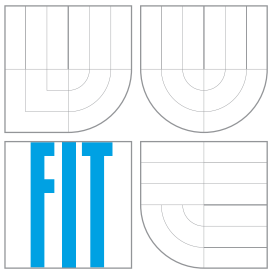
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROSTŘEDÍ PRO ZPRACOVÁNÍ DAT ZE ZACHYCENÉ SÍŤOVÉ KOMUNIKACE

FRAMEWORK FOR CAPTURED NETWORK COMMUNICATION PROCESSING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN PLUSKAL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ONDŘEJ RYŠAVÝ, Ph.D.

BRNO 2014

Abstrakt

Práce pojednává o možnostech získávání dat a jejich analýzy ze zachycené síťové komunikace. Jsou zhodnoceny možnosti aktuálně dostupných volných a proprietárních řešení jednotlivých nástrojů i celých prostředí určených pro síťovou forenzní analýzu. Provedením analýzy těchto nástrojů byly zjištěny nedostatky, pro které není možná integrace již hotových řešení pro záměry projektu SEC6NET, a dále byly stanoveny cíle, které navržené řešení musí splňovat. Na základě cílů a znalostí z předchozích prototypů řešení byla provedena dekompozice problému na jednotlivé funkčně související bloky, které byly implementovány jako nezávislé moduly schopny spolupráce. Správná funkcionalita je po každé změně v implementaci testována pomocí sad Unit testů, které pokrývají majoritní část kódu.

Před zahájením samotného vývoje bylo nutné zhodnotit aktuální situaci v komerčních i open-source sférách řešení. Srovnání nástrojů používaných pro forenzní síťovou analýzu (pojednání uvedeno v Kapitole 2) nám dalo jasnou představu, na kterou část trhu chce naše řešení směřovat a jaká funkčnost je v jednotlivých nástrojích nepříliš povedená. Následně byly stanoveny hlavní požadavky a směr, kterým by se měl vývoj ubírat.

Na začátku vývoje rekonstrukčního frameworku stála fáze vytvoření návrhu architektury a dekompozice průběhu zpracování zachycené komunikace do ucelených částí jednotlivých modulů (podrobně popsáno v Kapitole 3). Využití předchozích znalostí a zkušeností získaných vývojem rekonstrukčního nástroje Reconsuite nám pomohlo při formování fronty zpracování(pipeline), kterou budou data při zpracování procházet. Následně byly navrženy základní komponenty provádějící práci se zachycenou komunikací v různých formátech PCAP souborů, rozdělení komunikace na konverzace, provedení defragmentace na úrovni IP a v případě komunikace TCP provedení reassemblingu daných toků. V rané části vývoje jsme se zaměřili na komunikaci zapouzdřenou v nízkoúrovňových protokolech Ethernet, IPv4/IPv6, TCP a UDP.

Po definici rozhraní komponent bylo nutné provést další výzkum síťových protokolů a vytvoření algoritmů pro jejich zpracování ze zachycené komunikace, která se svým charakterem liší od standardní a není tedy možné ji zpracovávat dobře známými postupy z RFC či jader operačních systémů. Protože proces zpracování zachycených dat se na komunikaci přímo nepodílí, tak v případě, kdy dojde ke ztrátě či poškození při zachycení, nebo je komunikace směřována jinou cestou, atd., není možné data získat pomocí znovu zasílání, ale je nutné využít jiné mechanismy k označení či obnově takto chybějících dat. Prvotní návrh algoritmu provádějících IP defragmentaci a TCP reassembling je uveden v Kapitole 4. Po implementaci a otestování byl zjištěn problém se separací jednotlivých TCP toků (TCP sessions), který nebylo možné řešit původním návrhem. Po analýze tohoto problému byla změněna architektura procesní pipeline s výsledným zvýšením počtu rekonstruovaných dat v desítkách procent.

V závěrečné fázi je popsána metodologie jakou bylo porvedeno testování výkonu implementovaného řešení a srovnání s již existujícími nástroji vykonávajícími podobnou činnost. Protože rekonstrukce aplikačních dat je příliš specifická záležitost, při srovnání výkonu byla měřena rychlost zpracování a potřebná paměť pouze při provádění separace toků, IPv4 defragmentace a TCP reassemblingu, tedy operace společné pro všechny rekonstrukční nástroje. Srovnání ukázalo, že Netfox.Framework předčí své konkurenty Wireshark i Network monitor v rychlosti zpracování, tak v úspoře paměti. Jako testovací data byl použit jak generovaný provoz, který cílil na ověření výkonu v extrémních případech, tak i vzorky reálné komunikace zachycené v laboratorním prostředí. Detailnější analýza výkonu spolu se srovnáním jsou uvedeny v Kapitole 5.

Abstract

This thesis discusses network forensic data analysis possibilities, together with data mining methods to extract data from an intercepted communication. Applicability of commonly available (open-source and proprietary) tools and whole frameworks is evaluated and basic requirements for complex analysis tool are stated based on that review. Using experiences gained in the past experimentation with prototypes, functionally related components were designed based on a Divide and Conquer methodology. Components were implemented as autonomous modules that are able to cooperate each one with another. By committing series of tests some deficiencies were identified in processing of non-standard data captures that were fixed by improvement of reconstruction algorithms. The basic functionality of individual components are validated using UnitTests with more then major code coverage. Finally, the performance of capture processing was benchmarked and compared to similar oriented tools.

Klíčová slova

Zákonný odposlech, prostředí pro analýzu síťového provozu, Netfox.Framework, Netfox.Detective, NPlangCompiler, TCP znovu sestavení toků

Keywords

Lawful interception, network forensic framework analysis, Netfox.Framework, Netfox.Detective, NPlangCompiler, TCP reassembling

Citace

Jan Pluskal: Framework for Captured Network
Communication Processing, diplomová práce, Brno, FIT VUT v Brně, 2014

Framework for Captured Network Communication Processing

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Ryšavého, Ph.D.

.....
Jan Pluskal
May 26, 2014

Poděkování

Rád bych touto cestou poděkoval svému vedoucímu práce Ing. Ondřeji Ryšavému, Ph.D., za odbornou pomoc, cenné technické rady a důvěru spojenou s volností při zpracování této práce. Má vděčnost patří i všem kolegům: Bc. Martinu Marešovi, Ing. Vladimíru Veselému, Ing. Rudolfu Kajanovi, Ing. Michalu Zachariášovi z Rekonstrukční skupiny a dalším z projektu SEC6NET, kteří se mnou na vývoji Netfox.Framework spolupracovali, jejichž znalosti a osobní nasazení byly nepostradatelnou inspirací.

V neposlední řadě bych chtěl poděkovat své přítelkyni za její podporu, trpělivost a neomezenou zásobu vynikajících krekrů, kterými mě zásobila při psaní této práce.

Anzac biscuits

- 2 cups (180g) rolled oats
- 1 cup (150g) plain (all-purpose) flour
- 2/3 cup (150g) caster (superfine) sugar
- 3/4 cup (60g) desiccated coconut
- 1/3 cup (115g) golden syrup
- 125g unsalted butter
- 1 teaspoon bicarbonate of (baking) soda
- 2 tablespoons hot water

Preheat oven to 160°C (325°F). Place the oats, flour, sugar and coconut in a bowl and mix to combine. Place the golden syrup and butter in a saucepan over low heat and cook, stirring, until melted. Combine the bicarbonate of soda with the water and add to the butter mixture. Pour into the oat mixture and mix well to combine. Place tablespoonfuls of the mixture onto baking trays lined with non-stick baking paper and flatten to 7cm rounds, allowing room to spread. Bake for 8–10 minutes or until deep golden. Allow to cool on baking trays for 5 minutes before transferring to wire racks to cool completely. Makes 35.

Contents

Chapter 1

Introduction

The main goal of this thesis is to introduce, describe and document *Network forensics framework* which is part of an investigation intent the *NETwork FOrensic eXtendable analysis tool* (Netfox) of reconstruction group found under the SEC6NET grant at the *Faculty of Information Technology, Brno University of Technology*. The motivation behind the *Netfox.Framework* is a requirement to create a tool which will be used in a real investigation environment and must be adapted right on it's needs. That adaptation is the main reason why we have decided to create another framework in addition to either open source or proprietary solutions existing. Every and each one of these products has a potential and is really great in some subset of actions, but in global view there are always some circumstances which are so problematic to be bypassed that the product cannot be used for our intentions.

Our goal is to provide a complex but compact, integrated analysis tool which includes several analysis tasks at different levels. As an input for the analysis is a file with captured communication obtained on network active device near the suspect or the service that the suspect used or somewhere along the way. The quality of possibly reconstructed data is different on each point of capturing, therefore, each point is preferred for different use-case. For more details see chapter 4.

Based on a consideration that investigator has a collection of files with captured network traffic and his task is to gather relevant information or to reveal a suspect identity, traditional methods representing carving, drilling and search techniques are employed. The carving is representing a set of methods used to retrieve a specific information for example an email or IM communication. The drilling is performing a unveiling details about the specific information from the concise abstract provided by the carving. The search techniques are employing full text search on many levels of abstracting beginning with searching between packet payloads and finishing with a full text search on reconstructed objects.

On the investigator demands, there should be available a set of application analysis modules that might be applied to drill and export the vital information gathered during the carving represented by other modules. During the investigation process the investigator needs to have a meta-information about the capture file itself, for example size of capture, communicating parties, enumeration and distribution of application protocols. That information are vital for an investigator to apply his expert knowledge to decide which investigation technique to employ.

The *Netfox.Framework* was designed with a consideration of these investigation techniques and serves as a sophisticated analysis tool to help with network incidents detection, analysis, investigation and gathering evidence. The chapter 3 *Analysis and components*

design provides a light overview of designed framework structure (see Figure 3.2). All major components has been analyzed and their intended functionality discussed in dedicated sections. Besides a text description, several types of diagrams, providing support for better understanding of given issue, were used.

Netfox.Framework will be implemented as a *C#* projects compiled into dynamic libraries (assemblies), which will be published under the *MIT license* on the *Codeplex* project site SEC6NET 2014. The implementation is conceived like a test last development, because at the beginning, it is unknown if our concept will work on all intended application protocols. Because of that concern, the first version will implemented very swiftly like a proof of concept with a limited functionality and tested. Based on the test result, the framework will be extended with a new functionality (see chapter 4) and maintained as our primary reconstruction environment also intended as a testing ground for our new ideas and analysis approaches.

The last goal is to test the framework and perform comparison with other competing tools to demonstrate the speed of implementation of our unique design of capture processing pipeline. The preliminary results (see chapter 5) will focus on performance of low level components that are comparable to several existing tools to provide a general overview about the speed of capture file processing.

1.1 Netfox project rationale

The creation of *Netfox.Framework* was appointed by the *Reconstruction research group* leadership to develop our unique framework which will eliminate all disadvantages of current available solutions found in analysis state. Subsequently it has to provide an efficient source computing power utilization aimed on processing of large data and do that at in the real time application usage. This cannot be achieved on common computing equipment and according to our observation it would be impractical to process all data at once. For that, the common investigation should be factorized on several layers of abstraction in which top layers will provide the most general information overview in real-time. If the investigator is interested to use and deeply investigate the current data set specified by the top level information extracted in the most general pass then the framework will naturally provide more specific information gained by a deep packet analysis.

At the beginning of software development cycle, we have focused ourselves on the most crucial part and that is a creation of design of the *Netfox.Framework*. To be fully capable of designing, we have had to be familiarized with *Network forensics* practices methods of investigation of a real incidents. This topic is primary discussed in the chapter 2 *Network forensics framework analysis* along with the analysis of currently available solutions, and its features to inspire us in a process of requirements collection. Also, some basic knowledge base to protect against fundamental mistakes in development was founded. This part of analysis should have also demonstrated, if there were any solution that fitted our needs and requirements – also pointed out in this chapter. As concluded, there are non existing solutions suitable for re-using as a complex parts in *Netfox.Framework* .

The framework is a part of netfox family of tools developed by a *Reconstruction group*:

- NPlangCompiler – Ing. Ondřej Ryšavý, Ph.D.
- PmLib – Ing. Vladimír Veselý, Bc. Martin Mareš
- Detective – Bc. Martin Mareš
- ContentBrowser (Overseer) – Ing. Michal Zachariáš, Ing. Rudolf Kajan

Chapter 2

Network forensics

Let's assume that the term *Network forensics* might be a little bit confusing for many people (author included) and in a scientific literature several definitions could be found. Some of common definitions describe *Network forensics* as an upper layer or sub discipline of network (computer) security, which is focused on the investigation of a network incident threatening security of an organization or company. The *Network forensics* techniques are used to analyze an attack intercepted by the *Intrusion prevention system* (IPS). On the other hand if we look at the problem from a bigger perspective, the techniques of *Network forensics* can assist to reveal other types of *cyber criminality*, for example very common phenomenon on the internet deeply rooted in Czech society is a *violation of copyright law*. Other possible and more serious offence is a distribution of child pornography, which is spreading rapidly across the internet. Without *Network forensics* methods and the *Data retention law* (§88a z.c. 273/2012 Sb.) the criminologists would not have any way to fight it. This methods are also used to solve crimes with sexual based merits.

To be rigid, we must look in to commonly used definitions and try to find the best suited one related to this thesis. Security related one by Broucek and Turner 2001 claims that:

> *Network forensics* is not another term for network security. It is an extended phase of network security as the data for forensic analysis are collected from security products like firewalls and *intrusion detection systems*(IDS). The results of this data analysis are utilized for investigating the attacks. However, there may be certain crimes which do not breach network security policies but may be legally prosecutable.

These crimes can be handled only by *Network forensics* .

Accepting this it is important to be aware of another theory claimed by Berghel 2003.

> In *computer forensics*, an investigator and the hacker being investigated are at two different levels with the investigator at an advantage. In *Network forensics* the network investigator and the attacker are at the same skill level. The hacker uses a set of tools to launch the attack and the *Network forensics* specialist uses similar tools to investigate the attack.

Network forensics is defined in a similar way by Ranum 2013 as a capture, recording, and analysis of network events in order to discover the source of security attacks or other problematic incidents.

And also by Palmer 2001:

> *Network forensics* is a use of scientifically proven techniques to collect, fuse, identify, examine, correlate, analyze, and document digital evidence from multiple, actively processing and transmitting digital sources for the purpose of uncovering facts related to the planned intent, or measured success of unauthorized activities meant to disrupt, corrupt, and or compromise system components as well as providing information to assist in response to or recovery from these activities.

As we can see, around the *year 2000* the primary concern was network security defense of companies against attacks. This was before massive expansion of internet connectivity to almost every household and by this fact the massive growth of social networks.

Around *year 2010* the definition is slightly changing by adapting the law perspective by Pilli, Joshi, and Niyogi 2010.

> *Network forensics* is a natural extension of computer forensics. Computer forensics was introduced by law enforcement and has many guiding principles from the investigative methodology of judicial system. Computer forensics involves preservation, identification, extraction, documentation, and interpretation of computer data. *Network forensics* evolved as a response to the hacker community and involves capture, recording, and analysis of network events in order to discover the source of attacks.

For purposes of this thesis the definition by Pilli, Joshi, and Niyogi 2010 will be acknowledged and applied in a generalized point of view on detection of the most basic crimes committed by common people on the internet as summarized in the first paragraph.

2.1 Network forensic process models

From the time around *year 2001* when the network security ergo *Network forensics* starts to be a pressing matter due to the expansion of internet connection and when the first papers discussing it start to appear at the professional conferences, *Network forensics* begins to separate from the base discipline – *computer forensics*. This idea has been shown on *Digital Forensic Research Workshop* by Palmer 2001, presenting a model of framework in following steps: identification, preservation, collection, examination, analysis, presentation, and decision.

On the other hand, this and all models presented by Reith, Mandia, Casey and Palmer, Carrier and Spafford, Ciardhuain, Baryamureeba and Tushabe, Beebe and Clark were applicable to digital investigation and included the network forensics in a generalized form. The first real revolutionaries in this field were Ren and Jin 2005 who come forward with framework model in following steps: capture, copy, transfer, analysis, investigation and presentation, which is presented as a general model for *Network Forensics*.

Based on the general model and its predecessors, generic process model for the *network forensic analysis* (see Figure 2.1) were presented by Pilli, Joshi, and Niyogi 2010.

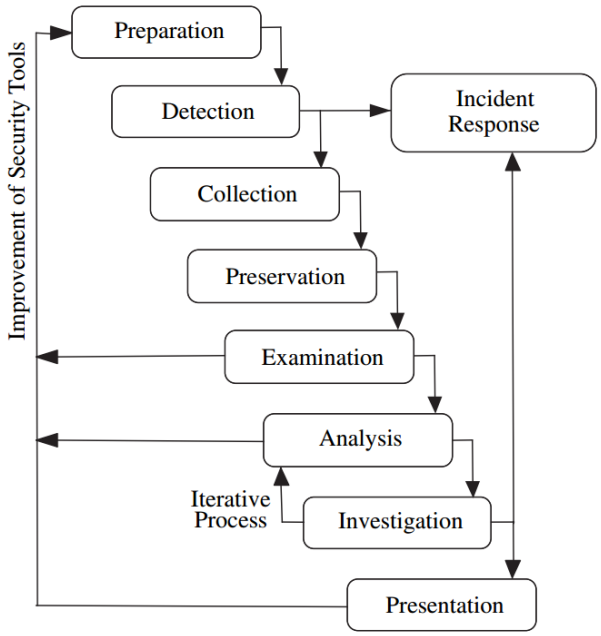


Figure 2.1: Generic process model for network forensics.

- *Preparation* – implementing a IDS/IPS, packet analyzers, firewalls, traffic flow measurement software and getting legal warrants
- *Detection* – reaction on alarms generated by security tools, determining nature of suspected attack and whether to continue or ignore alarm
- *Incident response* – reaction selected by the nature of attack, organization policy and legal and business constraints, planning future defense and recovery from damage, decide whether to continue investigation to gather more evidence
- *Collection* – acquiring data from traffic collectors, the goal is to provide maximum evidence with a minimum impact to the victim
- *Preservation* – export of collected data to read-only media, computing hash
- *Examination* – performed only on a copy of collected data, fuse data into one large data set, collecting evidence by methodical search of indicators of the crime
- *Analysis* – correlation of indicators to deduce important observations using the existing attack patterns, important parameters are related to network connection establishment, DNS queries, packet fragmentation, protocol and operating system fingerprinting
- *Investigation* – determining the path from a victim network or system through any intermediate systems and communication pathways, back to the point of attack origin
- *Presentation* – visualization of gathered evidence in an understandable language for legal personnel

2.2 Existing tools

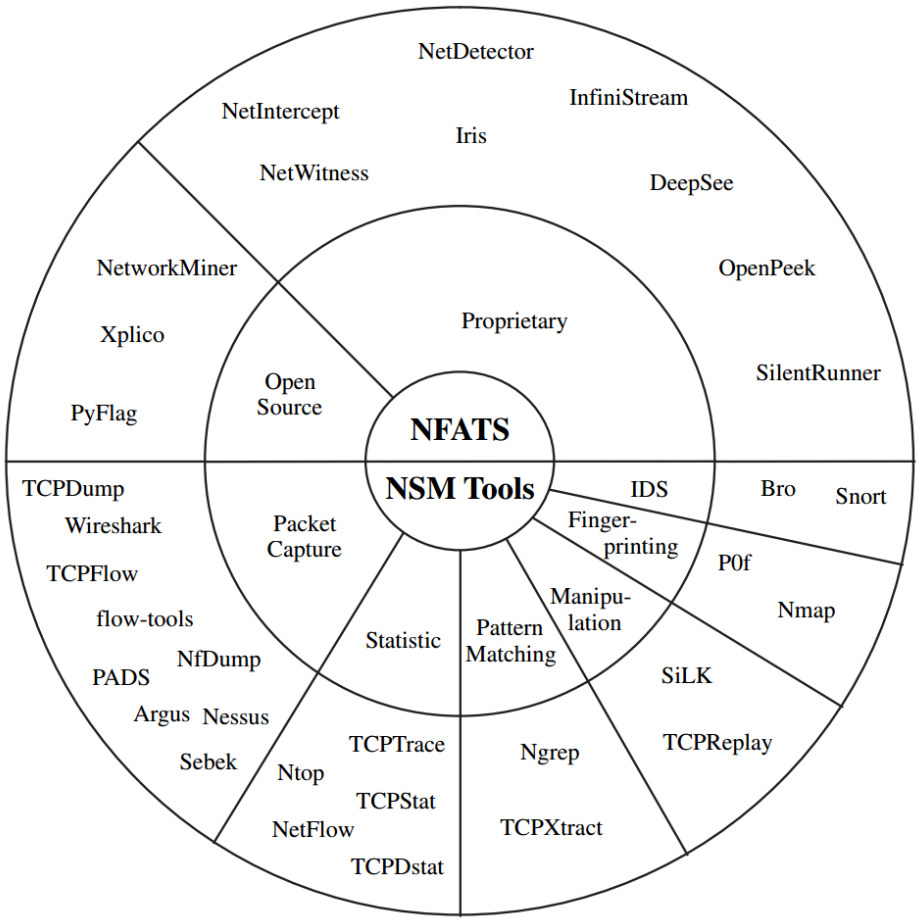


Figure 2.2: Classification of NFATs and NSM tools.

After we have agreed on what will be the meaning of the *Network forensics* term and the specified model of *Network forensics* in this thesis, it would be best to look at some existing frameworks and discuss pros and cons of solutions and why it would be impractical to use them and not develop our own better suited for our needs. A brief enumeration and description of *Network forensic analysis tools* (NFATs) will be also provided in Appendix B accompanied by the *Network security and monitoring* (NSM) tools in Appendix C. The graphical visualization is provided in Figure 2.2 where NFATs are grouped by its licenses and NSM Tools by usage.

Now when we have defined the *Generic process model for network forensic* (see Figure 2.1) it could be used as a base reference model for comparison with existing NFATs.

The first phase *Preparation* involves a placing of the network security and monitoring tools at strategic places for collecting evidence. This phase has to be done manually by adding a special hardware and configuration and cannot by bypassed as a software implementation. Maybe in the future with an evolution of *Software defined networking* it would be possible for NFATs to prepare this phase on dynamic basics due to its actual needs.

The NFATs (*NetIntercept*; *NetWitness*; *NetDetector*; *Iris*; *Infinistream*; *Solera DS 5150*; *OmniPeek*; *SilentRunner*; *NetworkMiner*; *Xplico*) work in all the other phases, ex-

Capability	Tcpflow	Wireshark	Net-Intercept	Net-Detetor	PyFlag
Tcpdump import and export	+	+	+	+	+
Flow/stream reconstruction	+	+	+	+	++
Protocol decoding	-	+	++	++	++
Data reduction	-	+	++	++	++
Known file detection/exclusion	-	-	-	-	+
Data recovery	-	+	++	++	++
Hidden data detection	-	-	+	-	++
Keyword searching	-	-	++	++	++
Audit log	-	-	-	+	+
Integrity checking mechanism	-	-	-	-	+
Loss documentation	-	-	-	+	+
Read-only collection	sd	sd	+	+	-
Read-only examination	-	-	+	+	+
Security	sd	sd	+	+	+

Note: Negative sign (-) when features are not implemented, a positive sign (+) when features are implemented, and a double positive sign (++) when features are implemented particularly well. Shortcut (sd) means system dependent.

Table 2.1: Rating NSM and NSATs tools by Eoghan and Casey 2004, *PyFlag* adopted from Cohen 2008.

cept for a few which are not applicable to *preservation* and *investigation* phases. *PyFlag* does not involve the *packet capture* and starts with the *examination* phase (Pilli, Joshi, and Niyogi 2010).

As you can see in *PyFlag*, the ability to capture and locally store a network traffic, implemented in early NFATs tools, is nowadays becoming obsolete, because with the evolution of IP networks using the Ethernet encapsulation on L2 with the transmission speed of a gigabit, since 1998 (Spurgeon 2000), cannot be captured on regular computer. Back then when *Network forensics* analysis was committed on a relatively small networks, where a connectivity to ISP was *Ethernet 10/100Mbit*, it was possible to use a single or small group of computers as probes to collect traffic. Nowadays, when we have a need to analyze a traffic of networks with a size of *100 gigabits*, it is completely impossible to accomplish this task on PCs so we need to use a special hardware equipment which stores captured traffic most often in *wireshark/TCPdump* form of PCAP file, but better in *PCAPng* which is able to provide more custom information about collected traffic like the probe id, some kind of the GUID specifying current collection set and other defined by user‘s needs.

As you can see in a comparison on the Table 2.1, the best NSATs seems to be the *PyFlag* which has several benefits in comparison to others. The biggest advantage is that it is *Open Source product* with *GPLv2 license* so it could be band to our needs but the product could not be sold as proprietary. *M. I. Cohen* claims in his paper „PyFlag – An advanced network forensic framework“ (Cohen 2008) that other commercial offerings include *Eeye‘s Iris* product as well as *Verint‘s data interception tools*. These tools specialize in analyzing network activity mainly for intrusion detection purposes. Although these tools claim the ability to extract documents contained within emails sent over the network, their integration with standard forensic techniques is limited (e.g., hash comparison, keyword

indexing and searching). These tools also do not offer the ability to integrate different sources of data into the same case, such as disk images and memory images.

PyFlag is based on concept of *virtual file system* (VFS). "The VFS is essentially a tree like structure which forms an arena for representing all objects within *PyFlag*. The VFS is modeled after a real file-system, and VFS objects are called *i-nodes*" (Cohen 2008). The biggest advantage of *PyFlag* is that its scanners – modules which purpose is to parse an object in VFS, produces objects that are also stored in VSF and are processed by other scanners. By this principle the *PyFlag* provides very efficient platform for committing *keyword search* which can found match, for an example in a DOC file compressed in a ZIP archive which is also compressed in a RAR archive and sent by an email to other user. This feature is very valuable when you need to process a very big amount of data in a relatively little time.

2.3 Challenges

The large number of security incidents affecting many organizations and increasing sophistication of these cyber attacks is the main driving force behind network forensics. The defensive approaches of network security like *firewalls* and *intrusion detection systems* can address attacks only from *prevention*, *detection* and *reaction* perspectives. The alternative approach of network forensics becomes important as it involves the *investigative component* as well (Almulhem and Traore 2005).

This *investigation component* of *Network forensics* is not crucial only for organizations, but for the law enforcement units and other agencies ensuring the state security as well. The generic process model is the same as provided in a Figure 2.1, but the *preparation*, *detection*, *incident response* is defined differently by the current situation.

For example, the police has a suspect and wants to gather all relevant evidence of committed crime. In this case *detection* and *preparation* phases are switched. *Detection* is reaction on some other subject like criminal charges. *Preparation* could by supplied be requesting court order to legalize suspect's network traffic interception. *Collection* is done on ISP's network without suspect's knowledge which is performed for a time period and followed with *examination* and *analysis*. These are performed offline on collected data. The rest phases are similar as defined in model description.

The other example are government's *law enforcement agencies*(LEAs). Their goal is to prevent a security threat and cannot wait for a long time to perform offline analysis. The threat is almost always imminent like a terrorist attack against an important summit or a visit of foreign politician. Their use-case correlates with a generic process model Figure 2.1 and performs online analysis on the data collected in a real-time. Most often, during the examination phase a pattern matching and a keyword search is performed to filter a very big amount of data to only few important, which are processed by humans in analysis phase. The incident response could be connected to all phases based on predefined triggers to accomplish the best possible reaction time to prevent the incident.

After the brief overview of existing tools the obvious question arises if we really do need to invent and implement a new network forensic framework, when there are to many existing solution that could be used. The main problem with the existing solutions is their primary focus to be used in corporate environment that has slightly different needs then LEAs have. Our goal and attention are focused on LEAs to provide the tool satisfying all their needs and requirements. To be able to decide how successful we were in accomplishing this task it is necessary to define global goals that we have to aim for (see on Table 2.2).

No.	Goal or requirement
1	Support common types of Link Layer technologies and encapsulation protocols, open and proprietary
2	Examination of all traffic even on non standard ports and encrypted traffic, all must be included in statistics and presented for analysis
2	Reconstruction of application protocol as many as possible, basic protocols like HTTP, POP3, SMTP, IMAP, IM's protocols are mandatory
3	Easy implementation of custom application protocol dissectors
4	Every reconstructed item must be able to trace back to its origin in capture file (files)
5	Traffic capture can be incomplete and missing data must be marked and if possible reconstructed from other packets (TCP retransmission, duplicates on Application Layer, etc.)
6	Traffic can be captured on multiple nodes in network at the same time so framework must be able to deal with many sources at once, frames can also be duplicated, missing and malformed
7	Process big data in captured files with appropriate time and memory space requirements
8	Reconstruction of encrypted data with provided decryption key

Table 2.2: Global goals and functionality requirements on *Netfox.Framework*

The whole framework shall be incorporated in to the *Examination* phase where it can satisfy all given requirements and provide an API for another applications to control its functionality and integrate it into themselves. This should be the biggest advantage compare to competing solutions, because this will provide you with an easy way to create your own *Network forensics* application without troubling your self with year of research and experiences in networking to build something that can be done once and then expanded to your needs.

Netfox.Framework will be completely modular, components will be separated and adding another application protocol dissector, recognizer, a link layer technology or anything else will be possible just by adding it into the processing pipeline in a prepared use-case or by creating your own use-cases and bending *Netfox.Framework* towards them.

Netfox.Framework will also be accompanied by another applications which will be based on it, use it as the background layer and work on upper model layers like *Analysis*, *Investigation* and *Presentation* phases. As an example of that application will serve the Netfox.Detective employing conventional analysis methods of drilling, carving and searching improved by a possibility of creation user plugins to satisfy a need for a unconventional investigation approach to by automated.

Chapter 3

Analysis and components design

According to needs and requirements presented in the *Motivation* section 2.3 it was decided to create a framework which would be modular and robust to enable a future expansion of functionality. The framework will be composed of modules, each of them will provide a limited functionality, performing only its single task. The data flow control will be provided by selected controller (managers) modules performing tasks needed to satisfy the use-case for which they were selected or it could be controlled by an external application. To write a such controller while not using one of prepared, the programmer has to have a detailed knowledge of the framework.

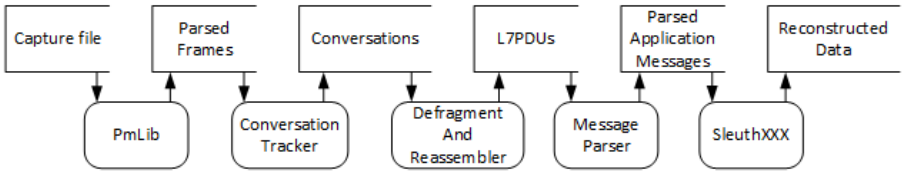


Figure 3.1: The data flow model diagram describing the *Netfox.Framework* in a general reconstruction process, the processing pipeline.

The *Netfox.Framework* is primary designed to be used for an application data reconstruction of captured internet traffic. The best way to describe a decomposition is to use a component diagram with a simple modules's description (see Figure 3.2) abstractly describing a control flow with connections between components starting in a *SleuthManager*. For a better understanding of the control flow, a quite abstract sequence diagram describing a use-case of application data reconstruction (shown in Appendix D) is provided.

The most accurate, but also the most complicated, is the last diagram freely modeling framework's components (see Figure 3.3) on many levels of abstraction. Diagrams are trying to point out the most crucial control sequences and connections. In the top left, the process of compilation NPL application protocol description by a NPlangCompile (see section 3.9) which is compiled to a *C#* class and used in separate *Sleuth* modules, is described. Sleuth modules are derived from the *SleuthBase* (see section 3.7) and serves as application protocol dissectors and exporters.

The process of data mining is controlled by the *SleuthManager* (see more details in section 3.1, see a data flow in Figure 3.1) which adds capture files using *CaptureManager* (see section 3.10) and initiates a process of conversation tracking and optionally a L7PDUs preparation. In a second phase, selected sleuths that should be used for a data mining are initiated and activated in a parallel evaluation.

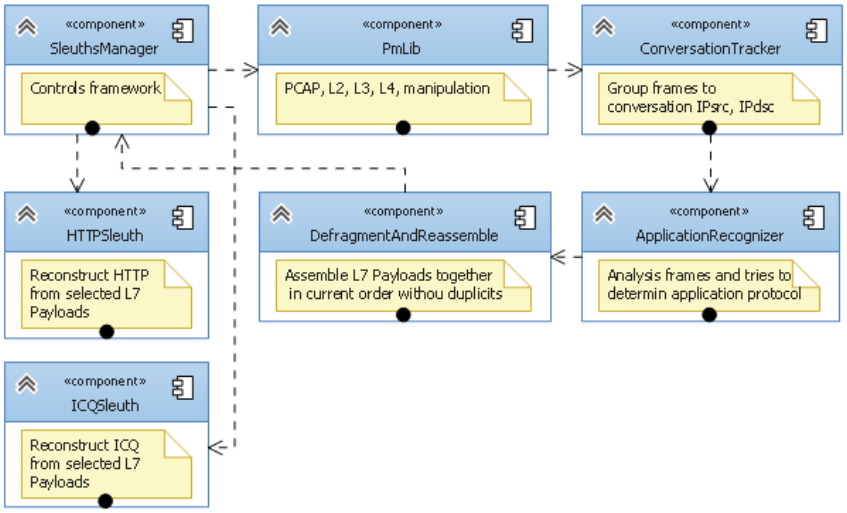


Figure 3.2: The component diagram describing the *Netfox.Framework* in a process of HTTP and ICQ data reconstruction from a captured traffic.

The sleuth loads conversations from the *ProcessingContext*, when activated, and uses the compiled protocol parser by NPlangCompiler to parse the input application layer data provided by the *SimplifiedMessageParser*. The data stored in capture files are accessed only by the PmLib and never directly.

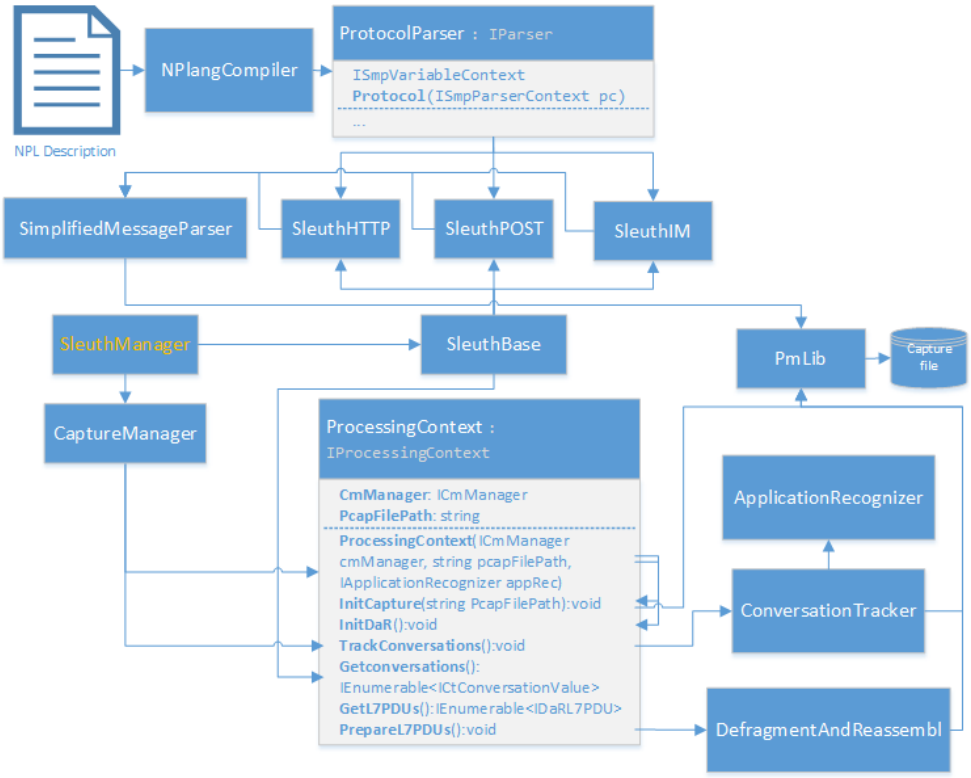


Figure 3.3: The diagram describing an abstract design of *Netfox.Framework*.

3.1 SleuthsManager

The *SleuthManager* is used when the investigation process is similar to the usage of reconstruction use-case presented in Figure 3.2. The module's goal is to control a data flow in the *Netfox.Framework* so appropriate modules will process data in a pre-selected order to accomplish a desired outcome. This module will provide an API to be used in the top layer application to hide the framework inner design from the application's business logic. The SleuthsManager's class diagram is provided on Figure 3.4.

The *SleuthManager* will control the framework (see in the Appendix D) by adding capture files (`AddCapture`) or just pre-selected conversations (`AddConversation`) to account. After that, when the input data set is ready, the framework activates sleuths (`ActivateSleuths`) – *sleuth* is a module providing data mining and exports results, in other words examination of application protocol.

The other feature of this component is to efficiently utilize a computing potential by load-balancing individual sleuths instances, run in threads on processor cores to provide the best performance on a current machine. The *SleuthManager* will run selected sleuths in parallel and orders them by their empirically determined weights, so the less efficient sleuth will be activated first, escalating into a minimization of a total running time.

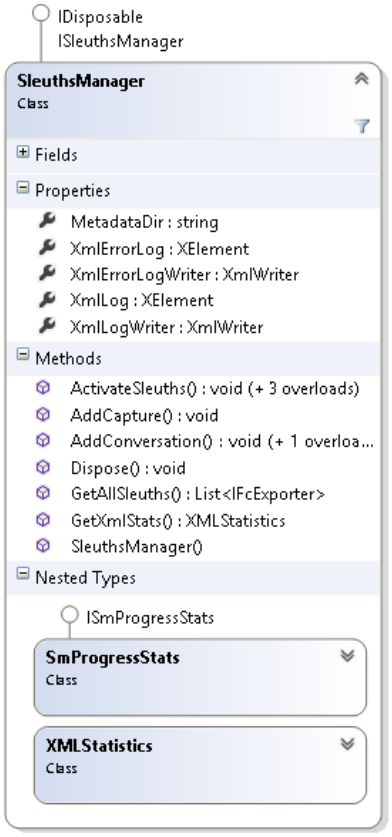


Figure 3.4: The class diagram describing an inner design of the *SleuthManager*.

3.2 PmLib

The PmLib is a capture file manipulation library written in *C#*, shielding upper layers from a manipulation with PCAP files and provides an *general interface* for that task. The library supports many capture file formats like *Wireshark/TCPDump's LibPcap* [1], *Microsoft network monitor's cap* [2] – version 2.0 and *PCAP-ng's pcap* [3] – version 1.0, shown in a Figure 3.5. The capture file is read using a C#'s *BinaryReader*, which is improving performance of the reading process.

The *PmLib* provides unified interface to access a parsed representation of frames stored in the capture file using the *IPmFrame* interface shown on Figure 3.6. Frames are parsed up to *transport layer* using *PacketDotNet's 0.13* [4] *dissectors* (Wikipedia 2012) supporting *Ethernet*, *LinuxSLL*, *PPP*, *PPPoE*, *IP*, *UDP*, *TCP*, etc... The *IPmFrame* inteface provides an access to only a small subset of parsed fields, which are mandatory for the *Netfox.Framework* to be able to process frames in the *ConversationTracker* and the *DefragmentAndReassemble* modules. This feature speeds up the whole processing, because there is no need to keep all capture files parsed up to *transport layer* in a memory, but only a small subset of them.

[1] http://www.tcpdump.org/
[2] http://www.scritub.com/limba/engleza/computers/Netmon-Capture-File-Format14459.php
[3] http://www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html
[4] http://sourceforge.net/projects/packetnet/

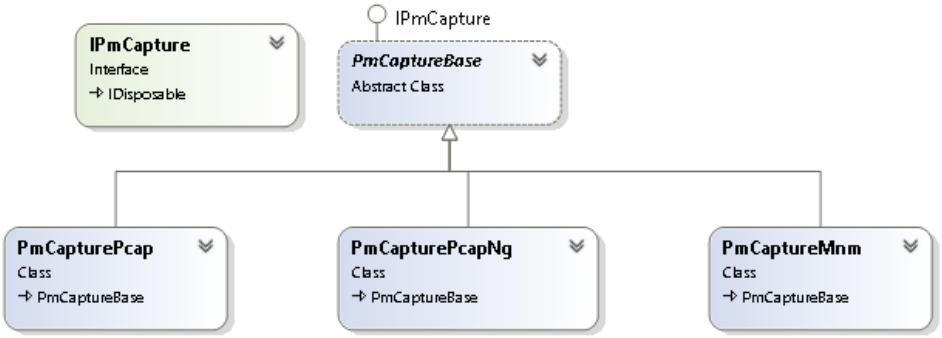


Figure 3.5: The class diagram clarifying an inheritance of the PmLib's *IPmCaptureXXX* differences for every captured file format.

For other use cases, two ways of accessing other fields are provided. The best practice is to use the *IPmPacket* encapsulating the original frame parsed by the *PacketDotNet* that provides more comfortable access to the often used fields. The other way is to use *PacketDotNet* parsed frame directly, but the manipulation with it requires more experience and produces a bigger code.

The another advantage lies in the *capture file indexing*. If the capture file was already parsed by the *PmLib*, the library creates an *index file* which contains among others a serialized list of instances of *PmFrames*. This feature ensures that the re-usage of the same capture file would not require to parse the whole file again, but only the index file, which by the performance measurement takes only about a 64.1% size of the original file stored on a hard-drive.

The PmLib provides also a support for *Application layer*(L7) PDUs thus the transport layer's payload retrieval based on the *ordered list of frames* as a Byte[]. For that to be possible, it is required to implement some mechanism that will provide the data stuffing for missing frames. That mechanism is called *VirtualFrames*. The *virtual frame* encapsulates an information about a space that needs to be filled. There are two possibilities how to fill missing data. The first is to use zeroes and the second is to generate some kind of noise with a selected noise provider.

3.3 ConversationTracker

The *ConversationTracker* separates frames according to their affiliation to a conversation. The conversation is understood as a pair of collections gathering *up* and *down flow frames*, ordered by *time-stamps*. Frames are separated to the flows based on an equivalence relation defined on theirs *IP Endpoints* and an equality of *IP protocol* type. The *IP Endpoint* shall be defined as a tuple containing an *IP address* and a *TCP/UDP port*.

One tracked conversation is stored as an instance of a *ConversationValue* object, shown on a Figure 3.7, that holds both up and down flows and an statistic information about the conversation.

The *ConversationTracker* is designed to hide its inner implementation against a programmer so that an access to a instance is provided only using an interface *ICtConversationTracker*. The *ConversationTracker* itself could be implemented using multiple types of data storage. The basic implementation will use a computer memory, because of its fast random access property and easy implementation supported by a programming language

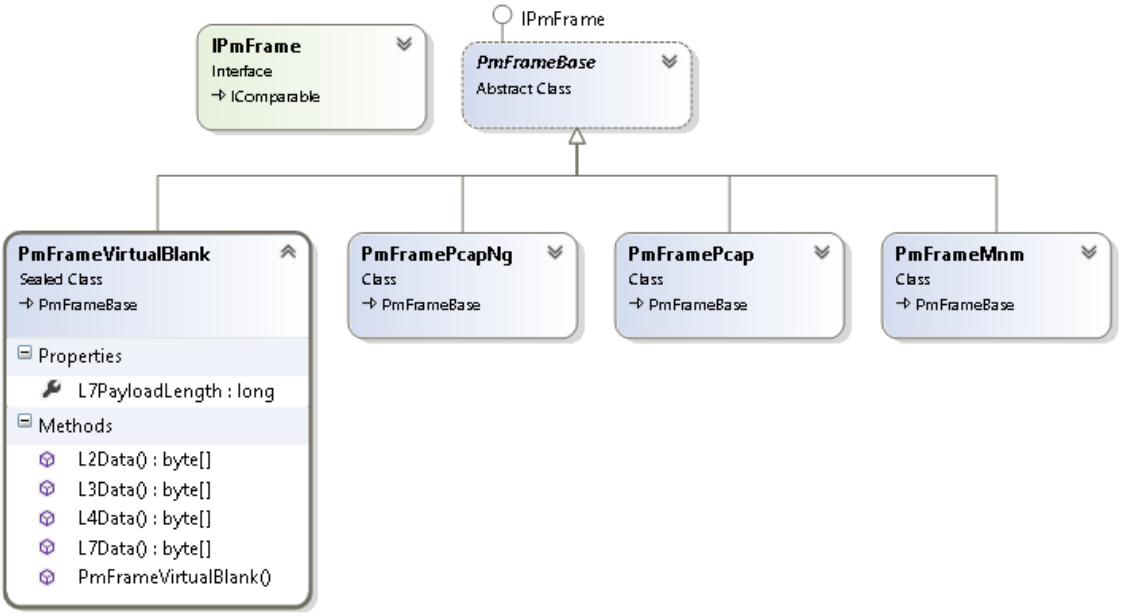


Figure 3.6: The class diagram clarifying an inheritance of the PmLib's *IPmFrameXXX* differences for every captured file format.

as a collection of *ConversationValue* objects. If need be, conversations could be stored in various types of databases. All that has to be done to accomplish this extension is to inherit the *CtConversationTrackerBase* and overload three basic manipulation methods **Creator** – creating a new *ConversationValue* object coresponding to a *ConversationKey*, **Provider** – generating a *ConversationKey* from a parsed packet – interfaced as *IPmFrame*, and **Updater** – updating given *ConversationValue* object using information from a newly parsed packet. The conversation tracking itself is initiated by a call of a *TrackConversation* method.

Besides them, the *ConversationTracker* provides two methods performing the conversation tracking – *ProcessPacketBase* and *ProcessPacketEvent*, which contains a build-in event that will be escalated with every newly found conversation. That way, programmer can subscribe a custom delegate to react on that event within the *ConversationTracker* identified by a *FcOperationContext*.

Thanks to the *ProcessingContext* present in all instances of the *ConversationValue* binding the conversation to its capture file and holding references to instances of all other *Netfox.Framework* modules, a method comfortably preparing application layer PDUs is also provided, just for that conversation to be used by a user later as desired. That method is called **PreparePDUs**– runs the *DefragmentAndReassemble* module.

Other methods of the *ConversationValue* will simply depend on the *ConversationTracker* and do not need any other modules. They are frame listing methods **GetFrames**, **GetDownFlowFrames** and **GetUpFlowFrames** – providing a list of *FrameNumbers* identifying frames in the capture file.

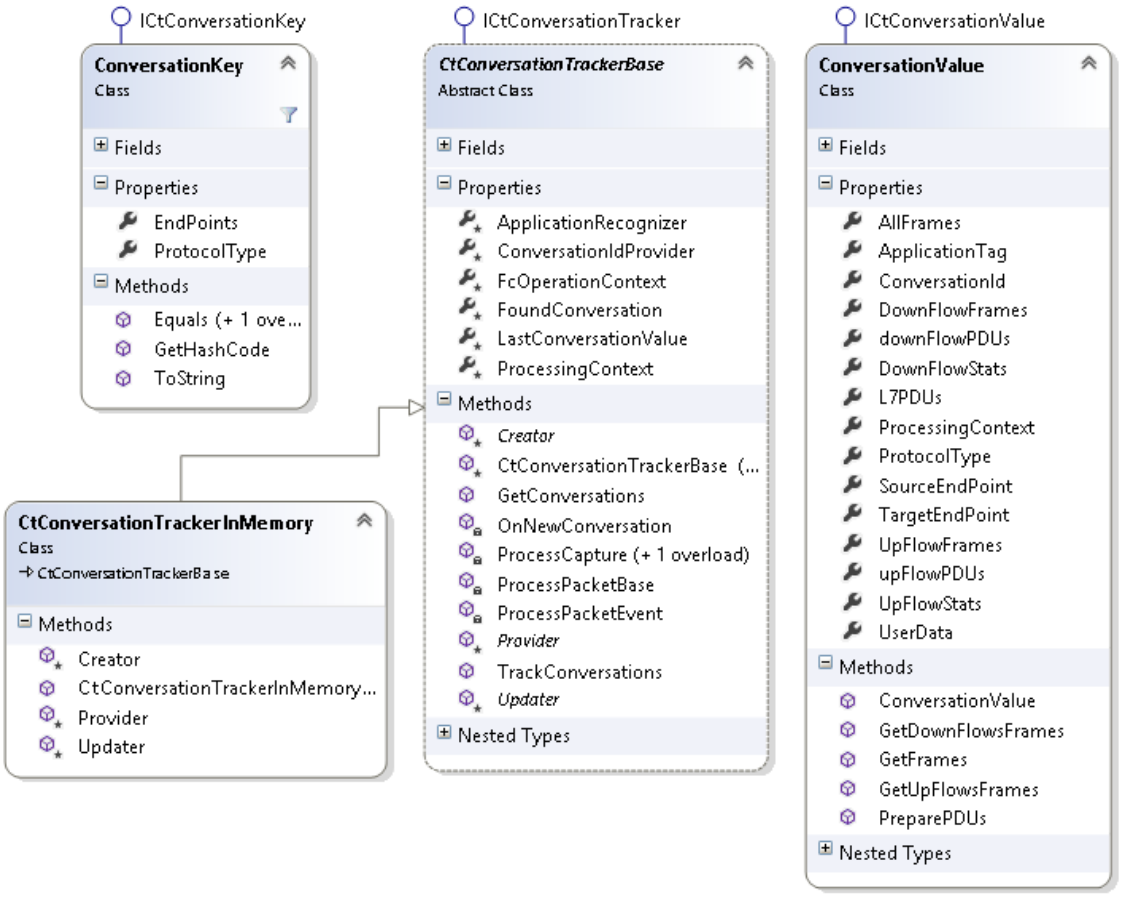


Figure 3.7: Class diagram clarifying inheritance and class members of *ConversationTrackr*.

3.4 ApplicationRecognizer

The *ApplicationRecognizer* module is responding to the user's demands to be able to identify an application protocol contained in the conversation. It is very probable that one conversation uses only one application layer protocol. Based on that assumption, several *application protocol recognizers* will be provided to suit actual needs of *Netfox.Framework* used. The *ApplicationRecognizer* module is designed to easily support addition of a programmer's custom recognizer just by an inheritance of *ApplicationRecognizerBase* class (shown in Figure 3.8).

The *ApplicationRecognizerBase* encapsulates a *KnownProtocolAndPorts* property (map<port,protocol>) which provides the programmer with a prioritized way how to force an *ApplicationRecognizer* to classify the traffic on selected ports with a mapped protocol.

The first and simplest recognizer classifies a traffic according to its transport protocol ports. There is no way how to determine in which traffic direction is a server and where is a client. For that reason, both source and destination transport layer ports are tested on an equality with ports contained by a port to protocol map. This module also contains a default port to protocol map with defined well known ports which can be redefined using a direct access to the public property *KnownProtocolAndPorts* or *CSV file* in a format see Listing 3.1.

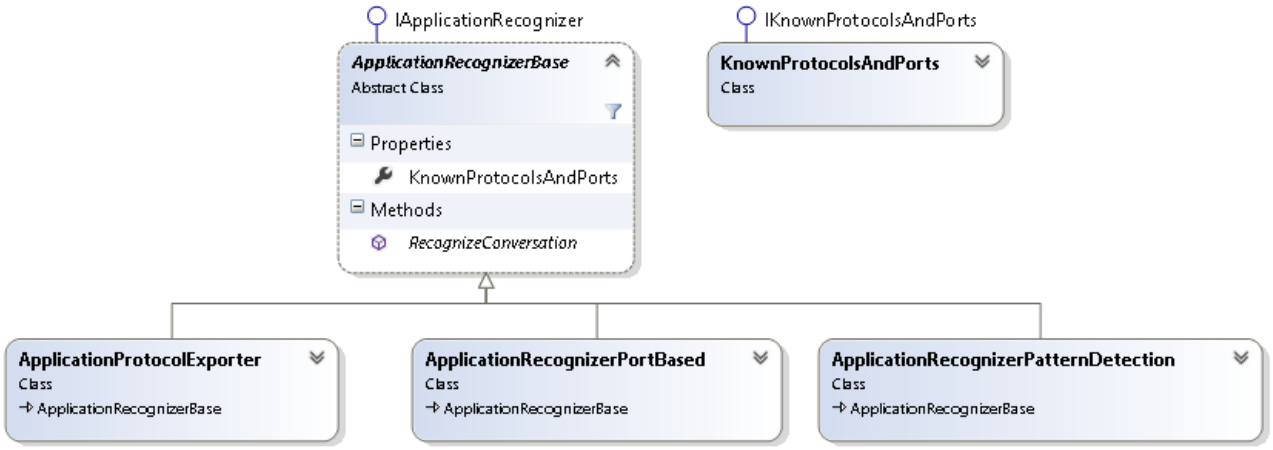


Figure 3.8: The class diagram clarifying an inheritance of *ApplicationRecognizers*.

```
1 Protocol1,1234
2 Protocol2,1235,1236
3 Protocol3,1237,12364,1239
```

Listing 3.1: Example of port to protocol mapping CSV configuration file.

The other inherited class will use some kind of protocol pattern analyzer to identify the text based application protocol by its typical identifiers. That could be for example a string *POST /test/demo_form.asp HTTP/1.1* that can be parsed using a regular expression and identified as HTTP if matched. For other protocols, binary based ones like ICQ (OSCAR protocol), the pattern matching cannot be applied for obvious reasons, but usually they have some special constant property that can be matched. For example, OSCAR contains in the first octet constant byte 0x2a.

The last one, the *ApplicationProtocolExporter* will be used to export application layer PDU for analysis by third party tool to determine the application protocol.

3.5 DefragmentAndReassemble

The *DefragmentAndReassemble* is the most important module in the *Netfox.Framework* . Without this module it would not be possible to reconstruct any data, because the data could be split into more frames and that frames could be also fragmented on routers along the way to destination side. The data could be captured as many frames containing TCP segments and even them could be fragmented to more frames containing IPv4 fragments. The frames could also get lost. In a case of UDP, the loss is definite. The TCP frame which is not delivered to the destination is re-transmitted. This is all related to a problem of transferring data on unreliable networks.

The sole purpose of this module is to reorder, filter and group frames to form ordered list containing frame numbers as their identifiers. This information is stored in a instance of *DaRL7PDU* that holds also statistics about success of the reassembling process. If some frames are missing, this anomaly is detected by the next mismatching calculated sequence number in the TCP frame. In a case of UDP, missing frames are not detected. Missing

frames are substituted with virtual frames provided by the *PmLib*. Every virtual frame has a unique frame number and carries information about the missing data length to be used as a padding. When the *PmLib* is called to retrieve the data from selected frames, the missing data length information is used to fill gaps.

The mechanism of *L7PDUs* were invented to bridge the gap caused by missing information about the application protocol messages segmentation in the TCP stream. The TCP hides the application protocol semantic, therefore, it do not carry information where in the stream the application message started and ended. The *L7PDUs* carries a part or whole application message and lets the application protocol parsed (knows the application protocol semantic) decide where the boundaries of single message are.

Frames are grouped in to *L7PDUs* by some basic rules, determined by experimenting and observing a behavior of *TCP stack* implementations in various operating systems. The common principle of all implementations is this:

- All TCP sessions begins with an exchange of frames with set TCP flags: *SYN*, *SYN+ACK*, *ACK*.
- All segments have a sequence number and a acknowledgement number. Numbers are incremented by special calculation with a length of frame. If some frames are missing, then their length could be recalculated.
- After every write into the socket at the application layer when the socket is flushed, the frame that carries the last data segment sets the *TCP PSH* flag.
- The *last segment* in TCP session sets one or combination of TCP flags *PSH*, *RST* and/or *FIN*.
- Every packet that is not acknowledged by a receiving side is re-transmitted. There are three reasons for that.
    1. The frame got lost or was malformed on the way between two routers, so it was discarded by router and never delivered.
    2. The frame was delivered, but not in a sufficient time.
    3. The frame was damaged and was discarded.

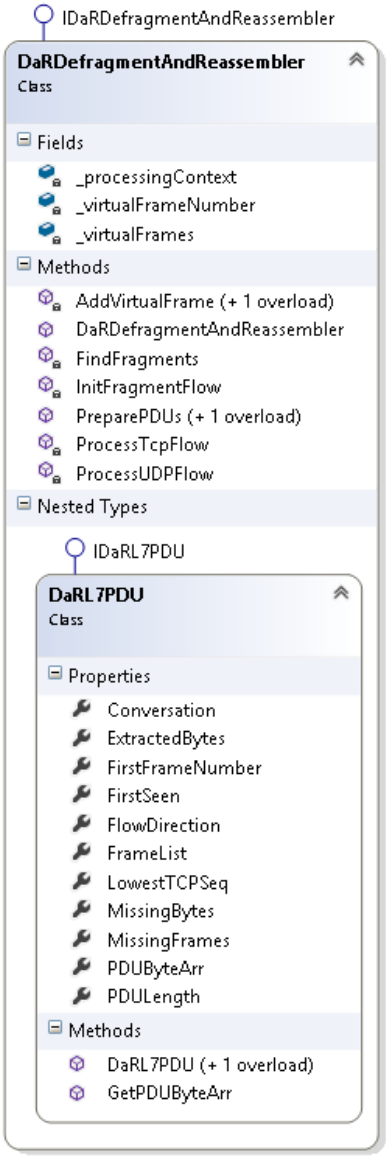


Figure 3.9: The class diagram describing an inner design of *DefragmentAndReassemble* module with a nested *DaRL7PDU* class.

The major problem of the TCP reassembling is that without a previous knowledge of *Application layer protocol* it cannot be determined that one complete application message is contained in one *L7PDU* or more. But it is positive that when *N L7PDUs* are grouped,

the last frame contains the end of the application message. This fact is used in a *PDU provider* defined in a *SimplifiedMessageParser* module (see section 3.6). It is also the base assumption on which the *Netfox.Framework* is built.

In a case that the UDP protocol is used on the transport layer, then PDUs are constructed from UDP data-grams in 1:1 ratio. That means that for one UDP data-gram one *L7PDU* containing that frame's number in a FrameList, is created.

3.6 SimplifiedMessageParser

The *SimplifiedMessageParser* behaves like an interpreter of compiled *Microsoft Network Monitor's NPL description* (Microsoft 2014) of the application protocol. The compilation is provided by a *NPlangCompiler* tool. Because a detailed programmer‘s specification of NPL is not provided by Microsoft, there is no guaranty that our understanding of NPL‘s user specification provided in a help file distributed with the Network Monitor will be totally correct. Besides, during a previous development of some NPLs that were not provided by Microsoft, several bugs were discovered in the Microsoft‘s proprietary implementation. We are intending to fix this bugs in *Netfox.Framework* but still be compatible with Microsoft‘s NPL implementation.

The NPL is a proprietary, imperative programming language that is a Turing complete, but not very comfortable for a programmer to use. The language was developed for describing a syntax of an application protocol to tell the parser how to process an input data. The NPL recognizes two types of variables. The first are *Fields* with declared data type, stored as a SmpVariableContext (see Figure 3.11) in a tree structure. Their location is by a fully qualified strings reflecting inner encapsulation. For example, a location of contact‘s status in a case of OSCAR IM protocol is qualified with „snac/snac011e/?/tlvrecordsnac011E/option/Status“, where ? substitutes for any item identifier. The second are *Properties* with a dynamic data type and limited validity to a *Global*, *Conversation* and a *Local scope* of parser. There is no assigning operator to assign a value to the field. The value is assigned when parsing process flow reaches the declaration of the field. The value assignment to the Property is valid only from:

- fields
- type casted fields
- type casted values started from a defined offset passed in a parameter of typecast
- plugins

There are several *Basic data types* and *Type casts* (see Table E.1 in appendix) that must be supported by *SimplifiedMessageParser*. There are also some Custom data types and type casts (see Table E.2 in appendix) intended for a simpler work with strings and data blobs. Sometimes, there is necessary to extend a functionality even more, therefore, there is a third element which are Plugins (see Table E.3 in appendix).

As an input for these sets of data types, casts and plugins a virtual buffer will be created, that will load *L7PDUs* in a hierarchical order by demands of the compiled application parser. By nature of the application protocol, there will be three data providers (see Figure 3.10) to fill this buffer. Each one of them provides PDUs ordered by a *timestamp* of first packet arrival time to a client.

- *BreakedInterlay* – when an application message is segmented to more PDUs, then the next PDU is selected from the same *flow direction* as the first PDU, but if next PDU in a row is not with the same *flow direction*, then buffer update fails.
- *ContinuingInterlay* – when an application message is segmented to more PDUs, then the next PDU is selected from the same *flow direction* as the first PDU and PDUs in other direction are skipped.
- *Mixed* – when an application message is segmented to more PDUs, then the next PDU is selected by the *timestamp* so it can be from the same or other *flow direction*.

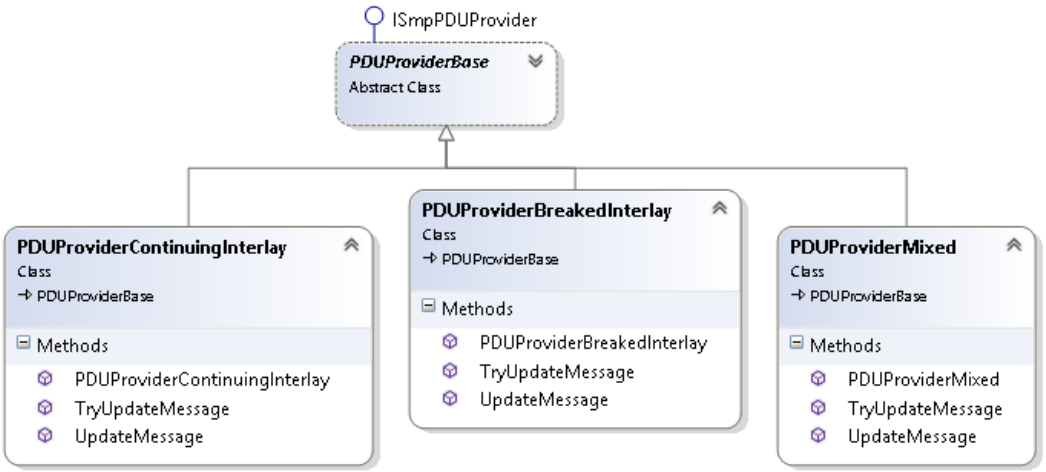


Figure 3.10: The class diagram clarifying an inheritance of PDU providers.

Every PDU provider model is best suited for a different application protocol behavior. For example, the email protocols *POP3*, *SMTP* and *IMAP* have no stop sequence to define where its data section ends so without the PDU provider *BreakedInterlay* it would not be possible to correctly extract data. For other protocols, that has no dependency on data received by the other side, the *ContinuingInterlay* PDU provider would be more suited since it would not stop when other data are received by the other side and it stops only when no other data are required by the application data parser. Finally, for protocols that has a strong dependency on received data the PDU provider *Mixed* is the most suited. *Mixed* provider provides both flows PDUs combined together and lets the application protocol parse to settle with them in its own direction.

It could be said that the *Mixed* PDU provider is the most dangerous one to work with. The application parser programmer has to have a complete knowledge about the application protocol. Otherwise if something goes wrong, parser might discard all data from the conversation without any warnings. On the other hand, the *BreakedInterlay* PDU provider is the safest one to use, because it cannot read after the point when another data is received by the other communicating side, therefore, it can discarded only a part of conversation by mistake.

The PDU provider is activated when parsing head reaches the end of virtual buffer, or some kind of peek type casts is called to require data that are missing from the buffer. In that case, the next PDU is selected using the specified nature of PDU provider discussed above.

When the *SimplifiedMessageParser* interprets a whole application protocol parser class, then several things are required:

- Evaluation of reading head position in the virtual buffer and determination if all loaded PDUs were also processed, if not, unprocessed PDUs must be return to conversation's unprocessed PDUs to be processed next time.
- Activate parsing method of *Sleuth* application parameterized with dynamic lists of *Fields* – containing all declared typed variables, *Properties* – contains dynamic variables acquired by some calculation on fields, type cast of field or Plugin usage, and *ParserContext* – provide information about parsing state, frames, etc. to *Sleuth*.
- Correctly finish and return control to the *Run* method of Sleuth when all PDUs are processed.

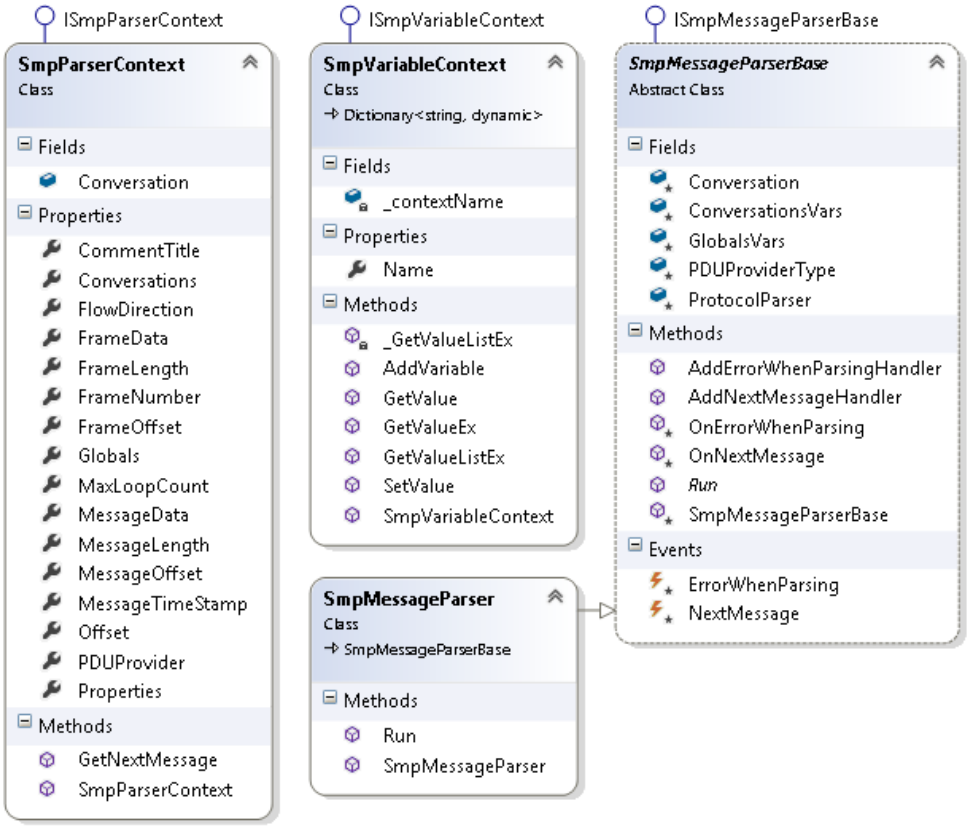


Figure 3.11: Class diagram clarifying a design of *SmpMessageParser*.

3.7 Application protocol Sleuths (HTTPSleuth, ICQSleuth, etc.)

Application protocol *Sleuths* are top level components that are aggregating information extracted of intercept communication on network parsed by *SimplifiedMessageParser* component described in section 3.6. The main functionality is to give semantic to syntactically parsed data. That means for one specific application protocol has to be one *Sleuth*, which understands protocol semantic and creates the added value by exportation of forensically significant data to XML representation data export log, which aggregates all data mined from current investigation.

This XML log (see Listing F.1 in appendix) and XML schema (seeListing F.2 in appendix) is meant to be input for Netfox.ContentBrowser.

Another new and flexible approach to the investigation by tool under development the *Netfox.Detective* is to increase granularity from capture files to conversations and direct control of framework to Detective tool. Also data mined by *Sleuths* will be encapsulated to the objects and directly forwarded to *Detective*, skipping the XML phase, which should increase performance and reaction time.

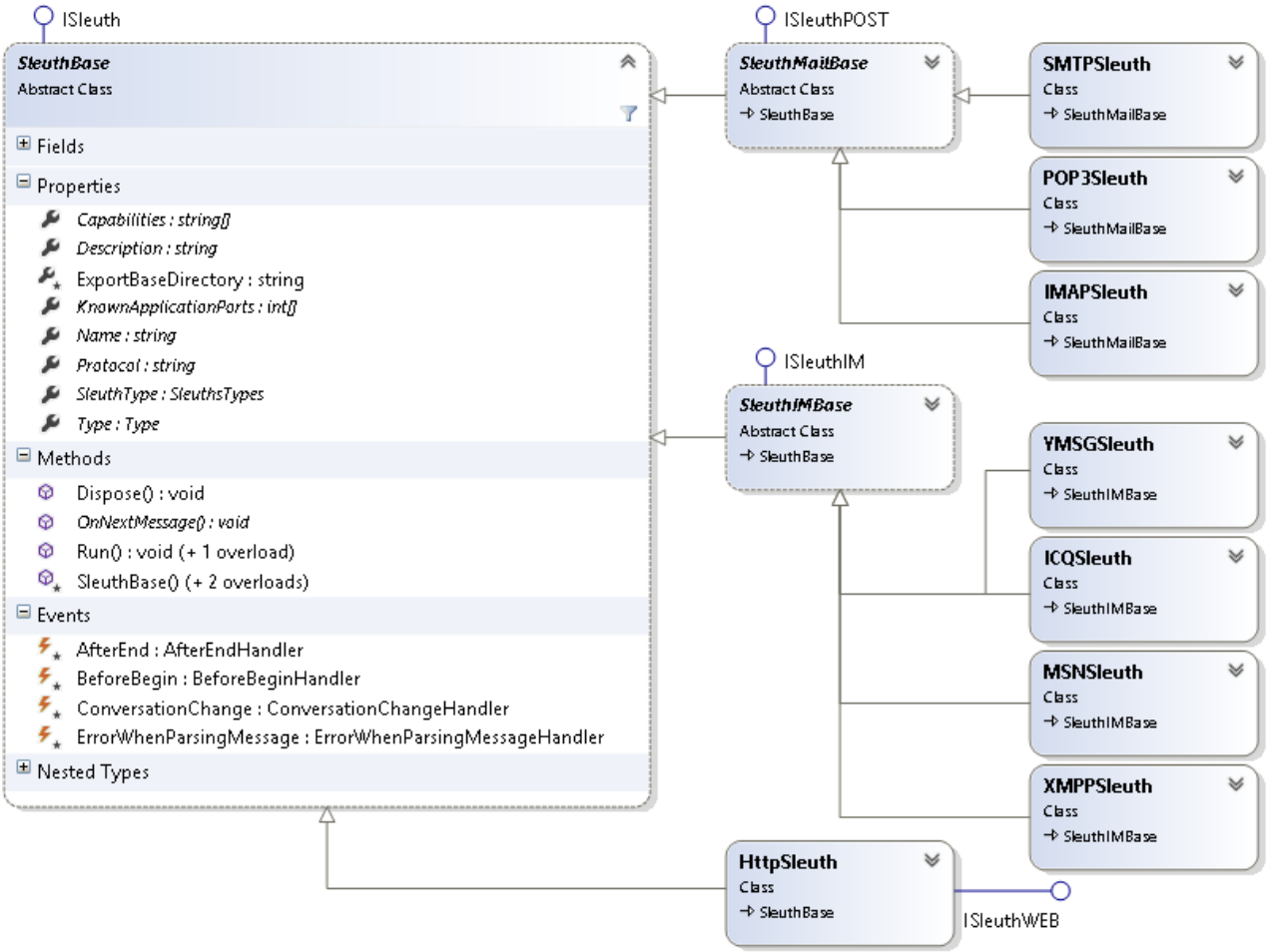


Figure 3.12: Class diagram clarifying a design of Sleuths.

The basic supported application protocols see on Figure 3.12 are *HTTP*, *IMAP*, *POP3*,

SMTP, *OSCAR*(AOL's ICQ), *YMSG* (Yahoo Messenger), *MSN*, *XMPP* (Jabber, GTalk). Output of all those *Sleuths* will be supported simultaneously by *ContentBrowser* and *Netfox.Detective*. Besides those native *Sleuths*, there can be easily created user defined sleuth using compiled NPL description of application protocol and inheriting *SleuthBase* class with override implementation of *OnNextMessage* method and set sleuth's properties. Activation and data export will be dealt by *Netfox.Framework* using reflection and inherited interface as a type determinant.

Sleuth module by itself must be re-entrant to ensure that can be run in parallel instances to assure the best performance available. Using *Netfox.Detective* or other custom tool to control framework processes could be achieved by mass parallel evaluation, possible because of refined granularity.

Sleuth is meant to be an autonomous unit that has no need to make any changes in the framework, like register itself, and it is able to configure its environment by applying settings stored in its properties, specifically *SimplifiedMessageParser's PDU provider mode*, *KnownApplicationPorts* for correct recognition of application tag in conversations by *ConversationRecognizer* and properties identifying *Sleuth* in *Netfox.Detective* like *Name*, *Description*, *Capabilities* and *Protocol*.

3.8 ProcessingContext

ProcessingContext is connection link, that will hold references to all instantiated low level objects that are required to processing one capture file. Class members are displayed on Figure 3.13. *Processing context* will be created, when a capture file is added to the *CaptureManager* module and will stay in the system the whole time until it's disposed and after that all references to all objects created during processing by framework shall be invalid. Objects required for low level processing ergo creation of *L7PDUs* are:

- *CmManager* as *ICmManager* – back reference
- *Capture* object as *IPmCapture*
- *CoversationTracker* as *ICtConversationTracker* with inner reference to used *ApplicationRecognizer*
- *DefragmentAndReassemble* as *IDaRDefragmentAndReassemble*

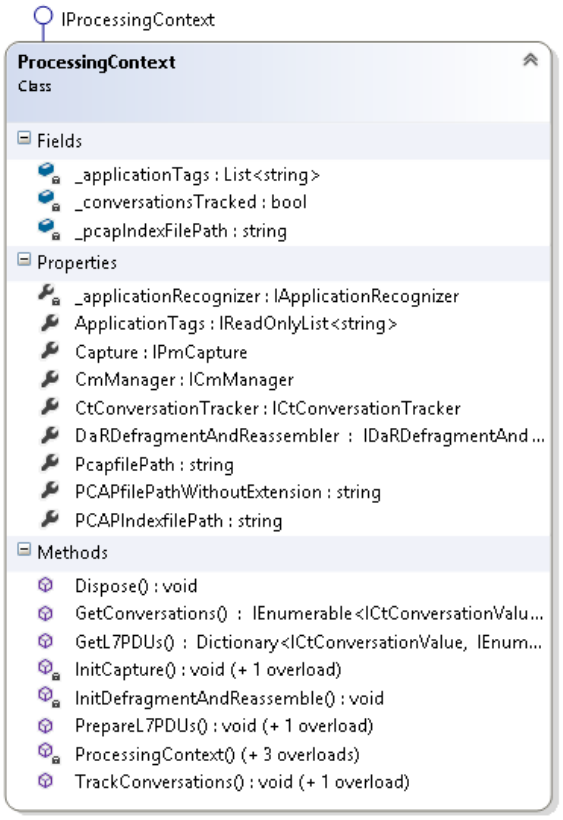


Figure 3.13: Class diagram describing inner members of *ProcessingContext* class.

Using *ProcessingContext's* member methods could be initiated a creation of *L7PDUs*:

- *TrackConversations*() – tracks UDP/TCP conversations and provides application tags
- *PrepareL7PDUS(ICtCoversationValue = null)* – creates *L7PDUS* for all/selected application tag

3.9 NPlangCompiler

For *Netfox.Framework* to be able to parse application layer it must provide syntax description of all wanted application protocols. To write such description for all protocols would be terrible waste of time, so we have looked for existing solution which serves our requirements and support a big amount of protocols. The wanted solution can already be found in application parsers written in proprietary *NPL language* used in *Microsoft Network Monitor*.

By default, *Network monitor* supports more then 384 protocols used on all layers of *TCP/IP model*. Using its parsers' editor, supporting validation and source code highlight, provides a way how to write and test user defined protocol parser on captured data sets.

NPlangCompiler is a console application that is able to generate *C#* class serving as an application protocol parser based on protocol's syntax description provided in *Microsoft Network Monitor's* NPL application parser.

Usage of *NPlangCompiler* is very simple. It takes a name of NPL description file as a parameter and generates the *C#* class with the same name and stores it in current working directory of command line. Integration of generated the *C#* class and *Netfox.Framework's Sleuths* is provided on Figure 3.14.

Version of *NPlangCompiler*, which generated directly the *C#* class described above, was meant to be a proof of concept and the future vision of this tool, that it would support more then just NPL protocol description and will compile it to intermediate code called a XPL, which is super-set of NPL providing abstract annotation of protocol. XPL constructions are defined by XML Schema and thanks to that, XPL is more suited for optimization like aggregation of unused fields, elimination of unused properties, etc. The reconstruction group has an ambition to develop, in time, a Graphical tool for manipulation with XPL and therefore provide an easy and fast way for users to develop their own application protocol parser.

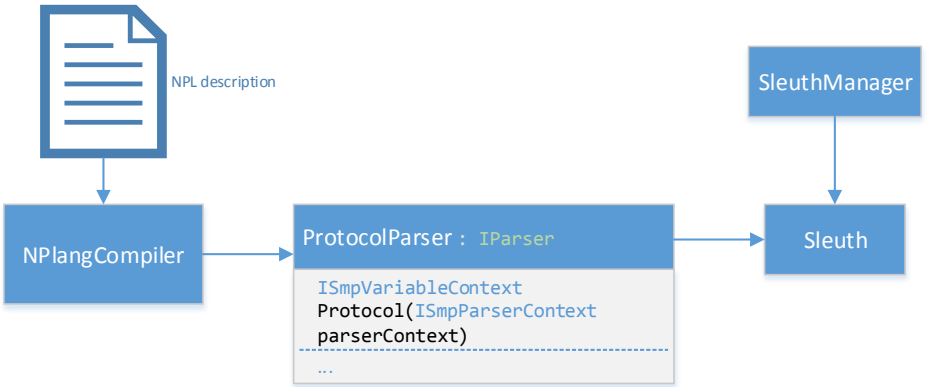


Figure 3.14: Class diagram clarifying a usage of NPlangCompiler.

3.10 CaptureManager

The *CaptureManager* serves as a storage of currently opened capture files, represented by *CmCaptureFile* objects, that could be reached in the framework environment. This module is nothing more then a wrapper of a *List* structure of *CmCaptureFile*. It might seen insignificant, but through its interface it provides a safer manipulation (*AddCapture*, *RemoveCapture*) with capture files and minimize a surface for errors.

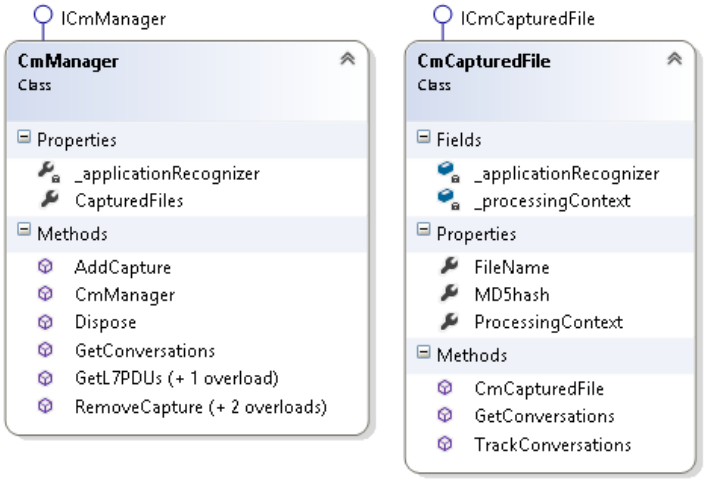


Figure 3.15: Class diagram clarifying a structure of CaptureManager.

The *CaptureManager* benefits from the fact that it is a single point that holds all references to opened capture files, therefore, it makes available a bulk evaluation of conversation tracking and retrieval of all conversations from all capture files (*GetConversations*) and/or retrieval of all L7PDUs from all conversations of all capture files (*GetL7PDUs*). Besides that, the *CaptureManager* provides default *ApplicationRecognizer* that can be used for all capture files in a session or override during addition of new capture file.

Chapter 4

Functional analysis of capture processing mechanism

Modules participating on the capture processing, like the *DefragmentAndReassemble*, are exceeding other parts of the processing pipeline by their complexity. Therefore, this chapter is dedicated to the capture processing mechanism, its functionality, implementation and a possible improvements to the design that mas made after the framework preliminary implementation. Because of that improvements, the *capture processing mechanism* had to be sightly changed and it is not implemented precisely as designed.

4.1 DefragmentAndReassemble module

To understand the purpose of this module it is mandatory to have a basic awareness of technologies used in a networking stack, specifically on the *Layer 3* (L3) – IPv4 (Wikipedia 2014c), IPv6 (Wikipedia 2014d) and the *Layer 4* (L4) – TCP (Wikipedia 2014e), UDP (Wikipedia 2014f). The *L3 fragmentation* and the *L4 segmentation* can occur simultaneously, but it could be processed in one pass.

4.1.1 IP fragmentation

The *Internet Protocol* (IP) implements a datagram fragmentation (Wikipedia 2014b), breaking segments into smaller pieces. Thanks to that, some frames may be fragmented to smaller ones that are able to pass through a link with a smaller *maximum transmission unit* (MTU) than was the original packet size. The *RFC 791* describes the procedure for the *IP fragmentation*, *transmission* and *reassembly* of datagrams. The *RFC 815* describes a *simplified reassembly algorithm*. The *Identification*, *Fragment offset* field along with *Don't Fragment* (DF) and *More Fragment* (MF) flags in the IP protocol header are used for a defragmentation of IP packets.

In a case, where a router receives a frame larger than the next hop's MTU, it has two options. If the internet protocol is IPv4: drop the PDU and send an Internet Control Message Protocol (ICMP) message, which indicates the condition *Packet too Big*, or fragment the IP packet and send it over the link with a smaller MTU. IPv6 hosts are required to determine the optimal Path MTU before sending packets. However, it is guaranteed that any IPv6 packet smaller than or equal to 1280 bytes must be deliverable without the need to use IPv6 fragmentation.

If a receiving host receives a fragmented IP packet, it has to defragment and pass it to the higher layer of networking stack. The defragmentation is intended to happen on the receiving host, but in a practice it may be done by an intermediate router. For example, network address translation (NAT) may need to defragment fragments in order to translate data streams, description provided in *RFC 2993*.

The IP fragmentation can cause excessive retransmissions when fragments encounter a packet loss and reliable protocol such as the TCP must retransmit all fragments in order to recover the loss of a single fragment. Therefore, sender typically uses two approaches to decide the size of IP datagram to send over the network. The first is that the sending host sends an IP datagram of equal size to a MTU of the first hop of a source destination pair. The second is to run the *path MTU discovery algorithm*, described in the *RFC 1191* to determine the path MTU between two IP hosts. As a result, the IP fragmentation can be avoided. This approach is not so reliable as it may seen, because there is no guaranty that during the conversation packets will be transmitted over the same routers and links.

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															

Figure 4.1: The IPv4 header, source (Wikipedia 2014c)

For the defragmentation purpose only a few fields from IPv4 headed are needed (see Figure 4.1): the *total Length*, *identification*, *MF flag* and *fragment offset*. Other fields are not important and does not need to be take into a consideration. More information with examples concerning fragmenatation could be found on Wikipedia 2014b. A receiver knows that the packet is a fragmented if at least one of the following conditions is true:

- The „more fragments“ flag is set. (This is true for all fragments except the last.)
- The „fragment offset“ field is non zero. (This is true for all fragments except the first.)

The receiver identifies matching fragments using the identification field. The receiver will use for defragmentation frames with the same identification field using both the fragment offset and the more fragments flag. When the receiver receives the last fragment (which has the „more fragments“ flag set to 0), it can calculate a length of the original data payload by multiplying the last fragment's offset by eight and adding the last fragment's data size. When the receiver has all fragments, it can put them in the correct order, by using their offsets. Then it can pass their data up the networking stack for further processing.

4.1.2 TCP Segmentation

The TCP provides a communication service at an intermediate level between an *application program* and the *Internet Protocol* (IP). That is, when an application program desires to send a large chunk of data across the Internet using the IP, instead of breaking the data into IP-sized pieces and issuing a series of IP requests, the software can issue a single request to the TCP and let it handle the IP details.

> IP works by exchanging pieces of information called packets. A packet is a sequence of octets (bytes) and consists of a header followed by a body. The header describes the packet's source, destination and control information. The body contains the data IP is transmitting.(Wikipedia 2014e)
>
> Due to network congestion, traffic load balancing, or other unpredictable network behavior, IP packets can be lost, duplicated, or delivered out of order. The TCP detects these problems, requests retransmission of lost data, rearranges out-of-order data, and even helps minimize a network congestion to reduce the occurrence of the other problems. Once the TCP receiver has reassembled the sequence of octets originally transmitted, it passes them to the receiving application. Thus, the TCP abstracts the application's communication from the underlying networking details.(Wikipedia 2014e)
>
> The TCP is a reliable stream delivery service that guarantees that all bytes received will be identical with bytes sent and in the correct order. Since packet transfer over many networks is not reliable, a technique known as positive *acknowledgment* with *retransmission* is used to guarantee *reliability of packet transfer*s. This fundamental technique requires the receiver to respond with an acknowledgement message as it receives the data. The sender keeps a record of each packet it sends. The sender also maintains a timer from when the packet was sent, and retransmits a packet if the timer expires before the message has been acknowledged. The timer is needed in case a packet gets lost or corrupted.(Wikipedia 2014e)

Offsets Octet		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

Figure 4.2: The TCP header, source (Wikipedia 2014e)

The Transmission Control Protocol accepts data from a data stream, divides it into chunks, and adds a TCP header creating a TCP segment. The TCP segment is then encapsulated into an Internet Protocol datagram, and exchanged

with peers. The term TCP packet, though sometimes informally used, is not in line with current terminology, where segment refers to the TCP Protocol Data Unit (PDU), datagram to the IP PDU and frame to the data link layer PDU. A TCP segment consists of a segment header (see Figure 4.2) and a data section. The TCP header contains 10 mandatory fields, and an optional extension field.(Wikipedia 2014e)

For reassembling will suffice only flags – SYN, ACK, PSH, RST, FIN, checksum, sequence, acknowledgement numbers and a windows size.

From a reconstruction point of view there is no need to follow a *TCP state diagram* (see Figure 4.3) and know in which state is a connection in every moment. If we try that, we have to realize that this task is not possible and reconstructed data would form only a fragment of original ones. The problem is in unreliable capturing device that not always stores all frames from network. For that reason the reassembling process has to be performed by a new algorithm that would be as context-free as possible to be able to reconstruct all data that are captured and fill missing ones with a stuffing.

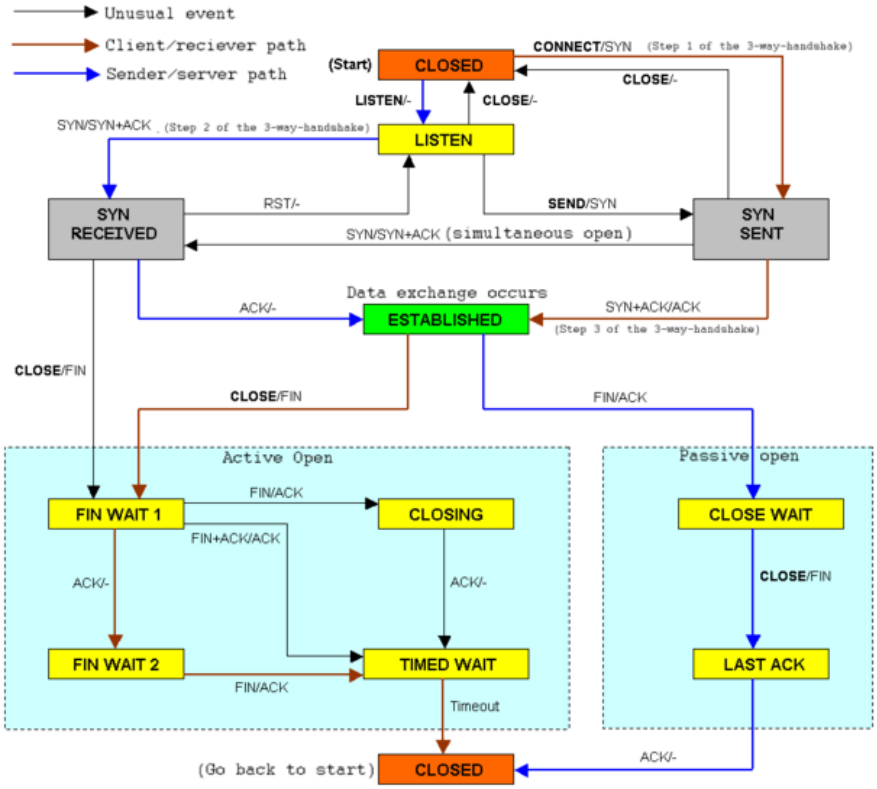


Figure 4.3: The TCP state diagram, source (Wikipedia 2014e)

As a reconstructing process sees it, the TCP communication could be divide into three parts.

- The initiation of a new connection (see Figure 4.4).
- The regular communication.
- The termination of communication (see Figure 4.4).

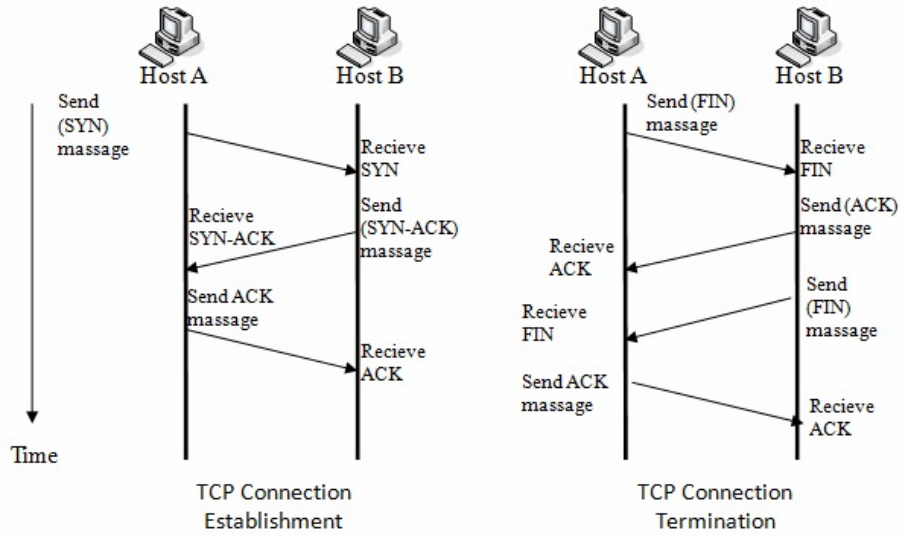


Figure 2.1. TCP session establishment and termination

Figure 4.4: The TCP initiation and termination sequence diagrams, source (Microtic 2010)

.

It is important to note, that *sequence numbers* (SEQ) are not only used as ordering index, but also carrying the information about the transferred data size. During initialization phase when the *SYN* flag is set, the *SEQ* number is incremented by 1 despite of the data part of the segment being empty. In regular communication state after the connection is established the *SEQ* number is incremented by the size of the TCP segment. This feature is enabling as to know how much data were lost and we can fill missing space by stuffing.

4.1.3 Simple DefragmentAndReassemble algorithm

After a brief introduction to the IP fragmentation and TCP segmentation problem, we can introduce a simple defragmentation algorithm. Our goal is to get the same data that was originally pushed to the networking stack by the application. After the understanding of the TCP we should be able to accomplished that. We could reorder packed by their *SEQ* numbers, calculate the *checksum* to remove invalids, filter duplicates and then finally fill out missing segments. That way we would get an correct output that could be provided for an example by the *TCPFlow* tool (Garfinkel 2014). The problem is that we would have missed the most significant meta information about the original data separation in the application messages. The reason, why this tool also does not provide this information is simple. Without a detailed semantic knowledge of application protocol it is not possible to strictly decide the application message boundaries in the reassembled flow.

To bypass this lack on information in the TCP, the algorithm that works with available data from TCP header, conversation statistics was developed in order to group reordered datagrams to data units called the *L7PDUs*. This units can contain a whole or a fragment of application message, but newer two or more application messages. Using this property in coordination with the *MessageParser* module we can achieve same results as we would have, if we had known the application protocol semantic at the time of reassembling. This is giving us an option of data preprocessing and lets us reuse them in more the one application data parser without the need of another reassembling.

Next description of the reassembling algorithm is related to the Listing 4.1. The reassembling process iterates through a collection of frames ordered by *TCP sequence numbers* (line 2). According to current and last *TCP sequence number* is computed a position alike indicator of current frame (variable x, line 4). If the x is *greater then zero* means that some frame or frames carrying between the last and current frame are missing, and needs to be filled with a *VirtualFrame* (line 7). Else if the x is *less then zero* (line 9) indicates that current frame might be a duplicate or retransmitte of previous frame, or TCP session closing frame with selected TCP flags *SYN*, *FIN* or *RST* (line 11). IF the current frame is duplicate, the *TCP checksum* has to be computed to eliminate possibility of frame corruption (line 11) and frame has to be exchanged with the previous in *currentL7PDU*. At last, the regular case if the x is equal to zero means that the frame is expected and if it carries some data (line 22) is added to *currentL7PDU* (line 24).

Each frame has to be checked for special signalizing properties of TCP. If the frame has set TCP flags *SYN*, *FIN* or *RST* (line 27) it is a sign of TCP hand shake or connection closure and *lastSeqenceNumber* has to be incremented by one, even thou the datagram carries no data. In other case, if the datagram carries some data, the *lastSeqenceNumber* has to be incremented by the size of that data (line 30).

The last tricky part is to decide the boundaries between L7PDUs (line 32-36). If the frame has set TCP flags *PSH*, *FIN*, *RST* and/or its payload is lesser then *conversationMTU* it is a signal that the current L7PDU is complete and new L7PDU object has to be created frames next to come (line 38-39).

This was an example of simple defragmentation algorithm that is in some form probably implemented in competitive tools. This algorithm do not take in account possible issues that might occur in the TCP session described in subsection 4.2.1.

```
var lastSequenceNumber = orderedConversation.First().TcpSequenceNumber;
foreach (var frame in orderedConversation)
{
   var x = frame.TcpSequenceNumber - lastSequenceNumber;
   if (x > 0) //some missing frames before this
   {
     AddVirtualFrame((uint)x, firstSeen);
   }
   else if (x < 0) //frame is retranssmitted
   { //in case that packet is TCPSyn retranssmit
     if (frame.TcpFSyn || frame.TcpFFin || frame.TcpFRst)
       continue;
     //in case of retransmission must be computed TCP Checksum
     if (frame.IsValidChecksum){
       currentL7PDU.RemoveDuplicit(frame);
       if (frame.Ipv4FMf) FindFragments(frame);
       currentL7PDU.Add(frame);
     }
     else
       continue; //if checksum is not valid frame is skipped
   }
   if (frame.L7PayloadLength > 0) { //regular frame in order
     if (frame.Ipv4FMf) FindFragments(frame);
     currentL7PDU.Add(frame);
   }

   if (frame.TcpFSyn || frame.TcpFFin || frame.TcpFRst)
     lastSequenceNumber++; // SYN increments SEQ
   else if (frame.L7PayloadLength > 0)
     lastSequenceNumber = frame.TcpSequenceNumber + frame.L7PayloadLength;
   //payload is smaller then MTU, means that it's not fragmented
   if ((frame.TcpFPsh || //END of L4PDU
        frame.TcpFRst ||
        frame.TcpFFin ||
        frame.L7PayloadLength < conversationMTU))
        && currentPDU.Count != 0)
        {
        flowL7PDUs.Add(currentPDU);
        currentPDU = new DaRL7PDU(conversation,flowDirection);
        }
}
```

Listing 4.1: The reassembling algorithm in pseudocode.

4.2 Conversation tracking deficiency

The original conversation separation approach considering every packet, that has the same key, belonging to the conversation was found as problematic. The original key as stated in the section about the design of the *ConversationTacker* module (see section 3.3) is insufficient to discriminate two or more application session from each other. When we consider that the *Network address translation* (NAT) (see Egevang and Francis 1994) is typically applied in the network, there could be several transport protocol sessions, during a time period of data capturing, with the same key and each of them could transport a different application protocol and belong to a different user.

In a case that the transport protocol is the TCP, a situation get even more serious because of virtual frames. When there would be two TCP sessions both with the same key. The gap inside their first and/or last sequence number will create a virtual frame filling a non-existing missing space.

For an example, there is a need to intercept the communication of suspicious service hosted at some data-center. The data-center has a redundant load-balancing network infrastructure that requires the traffic capturing to be employed on multiple active network devices at once. The service by its design communicates very often, transfers a big amount of data and for every data transfer opens a new TCP session, which expires, when the transfer is complete. The data-center uses NAT for traffic load balancing. After a certain time period, the load balancing NAT will reuse the same port for a new connection and after a longer time period the service will use randomly selected sequence numbers for newly initiated TCP session same as that which have been used for some previous session. From the services‘ point of view it is a correct behavior, because the previous connection was closed and the TCP session was newly initiate. But from the reconstruction point of view, communicating parties have the same source, destination endpoints and transport protocol, therefore, it considers communicating parties to be the same, nevertheless it might be two different users connecting the service.

There are also some specifics related to the captured communication which differs from the normal one. There is always a chance that the frame is delivered to the communicating side but is not captured. Therefore, when the captured communication is analyzed, the investigation can never be sure if the frame got lost only when capturing or even on real network. On the contrary, there could be a situation that the frame is captured but in a reality is never delivered and gets lost in a further parts of network.

Based on the knowledge, gained by experimenting with the *ConversationTracker* and the *DefragmentAndReassembler* modules‘ implementation using the original design was decided that design has to be change to include another mechanism to group frames to conversations. What was previously taken for the conversation will be from now on considered as *Bidirectional flow* and frames will be separated to the *conversations* in the time of TCP reassembling and/or UDP processing using various heuristics.

4.2.1 TCP session separation issues

The normal, completely captured communication (see Figure 4.5) can be reassembled without any difficulties, because packets could be ordered chronologically by TCP *sequence numbers*. After that, the reassembling process just concatenates L4 payloads into *TCP sessions*. If we assume that the flow starts with a set *SYN* flag and ends with set *FIN*, one TCP session is equal to that flow and also there are no valid data transmitted before the

SYN and after the *FIN* flags. We are able to detect that some data parts are missing by calculation and checking TCP *SEQ* numbers.

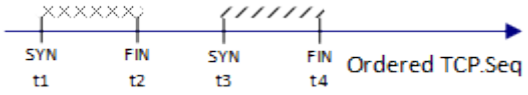


Figure 4.5: The normal, fully captured flow.

There are some major problems that must be dealt with. With the capture unreliability, there is possibility that the frame signalizing the beginning or the end of the TCP session is not captured (see Figure 4.6). In this case, a heuristic must be applied considering a timeout when the OS declare the TCP session invalid, or there is *SYN* flag set in the next frame that signalizes the *new TCP session*.

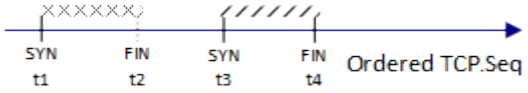


Figure 4.6: The missing datagram with a set *FIN* in captured flow.

In a case that the transferred data are big, there is a probability of the TCP *SEQ* number counter overflowing and the later-sent data appearing before the TCP session beginning, when the data-grams are being ordered by TCP *SEQ* numbers (see Figure 4.7). The TCP *SEQ* overflowing can happen together with a missing frame carrying the set *SYN* flag or data. Without it, we have no connection between this two TCP data sequences and it cannot be determined if it was one or two TCP sessions.

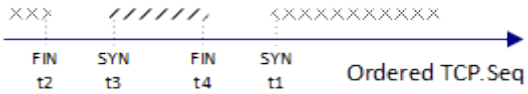


Figure 4.7: TCP Seq numbers overflow.

If the data have been captured over a long time period, there is a chance that TCP *SEQ* numbers could overlap (see Figure 4.8), and another heuristic has to be applied to discriminate frames that do not fit the TCP sliding windows using their time stamps.

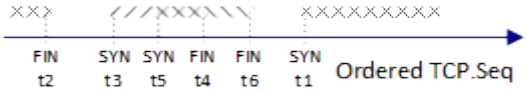


Figure 4.8: TCP *SEQ* numbers overlapping.

4.2.2 Redesigned capture processing mechanism

As soon as the mistake in the design of the conversation separation mechanism was discovered, it was obvious that the whole concept of capture processing has to be redesigned to be more flexible and reflect real conversations in captured data.

Same as the whole *Netfox.Framework*, the capture processing is also driven in a way of *lazy evaluation* or the *delayed evaluation* methodology. The processing is divided into several steps that has to be initiated sequentially if a result of the upper step is needed.

- The addition of the capture file to the *Netfox.Framework*
- The opening of *capture file* with indexation
- The initiation of the *bidirectional flow* tracking
- The initiation of the *conversation tracking* with:
  - The *IP defragmentation*
  - The *TCP reassembling*
  - The application protocol's *PDUs separation*
  - The *application protocol recognition*

New redesigned *Netfox.Framework* is presented on Figure 4.9.

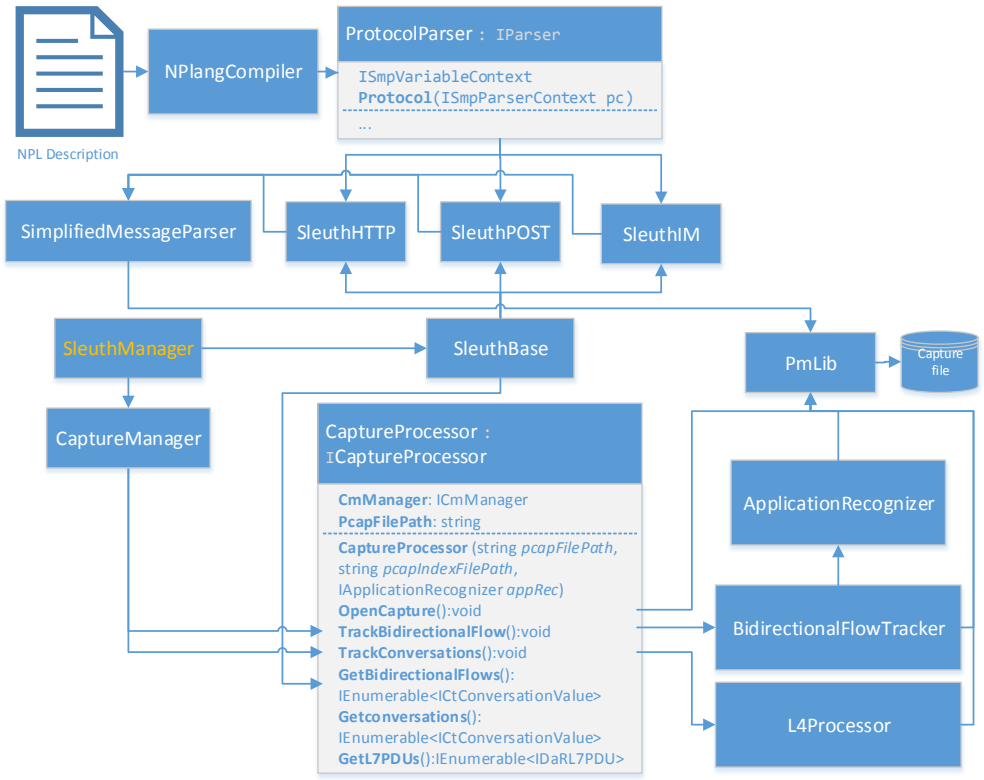


Figure 4.9: The diagram describing the abstract of redesigned *Netfox.Framework*.

CaptureProcessor

The *ProcessingContext* class which have glued all modules together was substituted by the *CaptureProcessor* class that have almost the same functionality, but divides so called conversation into two entities, bidirectional flows and transport protocol sessions more strictly specified. The *transport protocol session* shall be called the *conversation* from now on.

The *CaptureProcessor*, like the *ProcessingContext*, still represents one capture file's processing state in the framework and provides an comfortable interface for the capture file manipulation by the *Netfox.Framework*.

The *CaptureProcessor* implements methods to support all capture processing pipeline manipulation and result returning. Safety mechanisms that are preventing the user from getting results without previously initiation necessary operation and blocking double initiation of the same operation are also implemented.

The *CaptureProcessor* is also intended as a capture file identifier when more then one capture is processed at the same time in cooperation with other applications through the *FrameworkController* module.

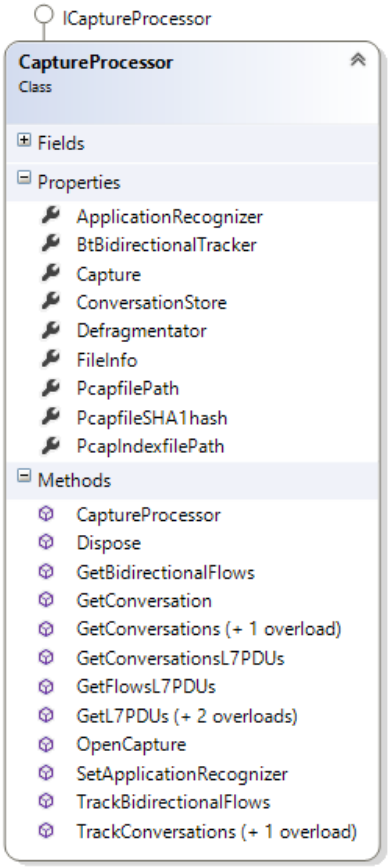


Figure 4.10: The class diagram describing inner members of *CaptureProcessor* class.

BidirectionalFlowTracker

The *tracking of bidirectional flows* is very similar to the previous version of the *conversation tracking* with some slight differences mentioned bellow (see Figure 4.11). The base concept of frame separation by their properties is left the same using structure *BtBidirectionalKey* (see Figure 4.12) for equality comparison.

Bidirectional flows are not exported during the tracking, but provided as a *IEnumerable* collection of tracked flows. This approach was chosen because there is no way to know if the flow contains all frames in given capture file before the bidirectional flow tracking is finished, ergo all frames are processed.

Another difference is that no application protocol recognition is applied in this state because one bidirectional flow might contains more application protocols. Therefore, the application recognition was pulled into upper layer in the capture processing pipeline the phase after the conversation separation.

The *BidirectionalFlow* value provides an aggregated view over properties contained in the belonging conversation values. For detailed enumeration of these properties see Figure 4.12. Some of these properties, which are necessary for future processing like the *BidirectionalFlowMTU*, *(L2|L3|L4)ProtocolType*,

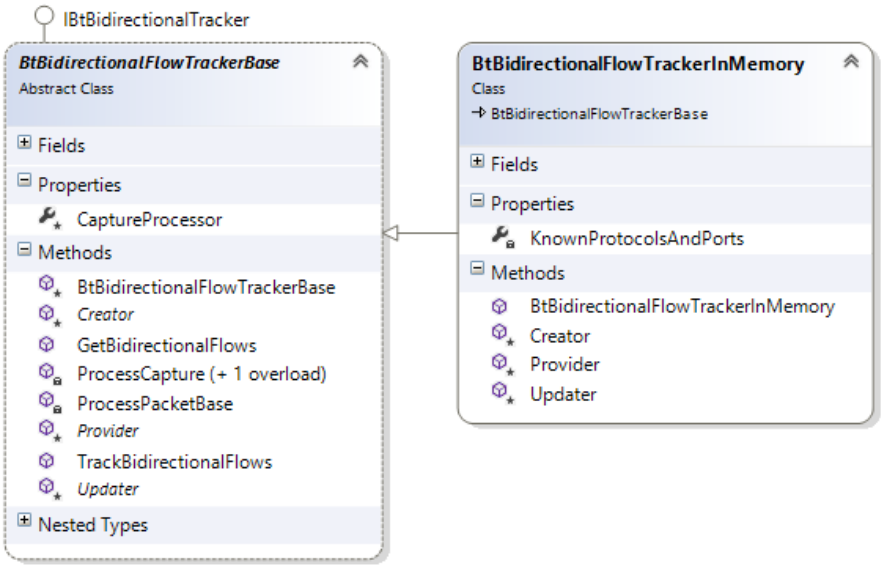


Figure 4.11: Class diagrams describing inner members of BidirectionalFlow tracking classes.

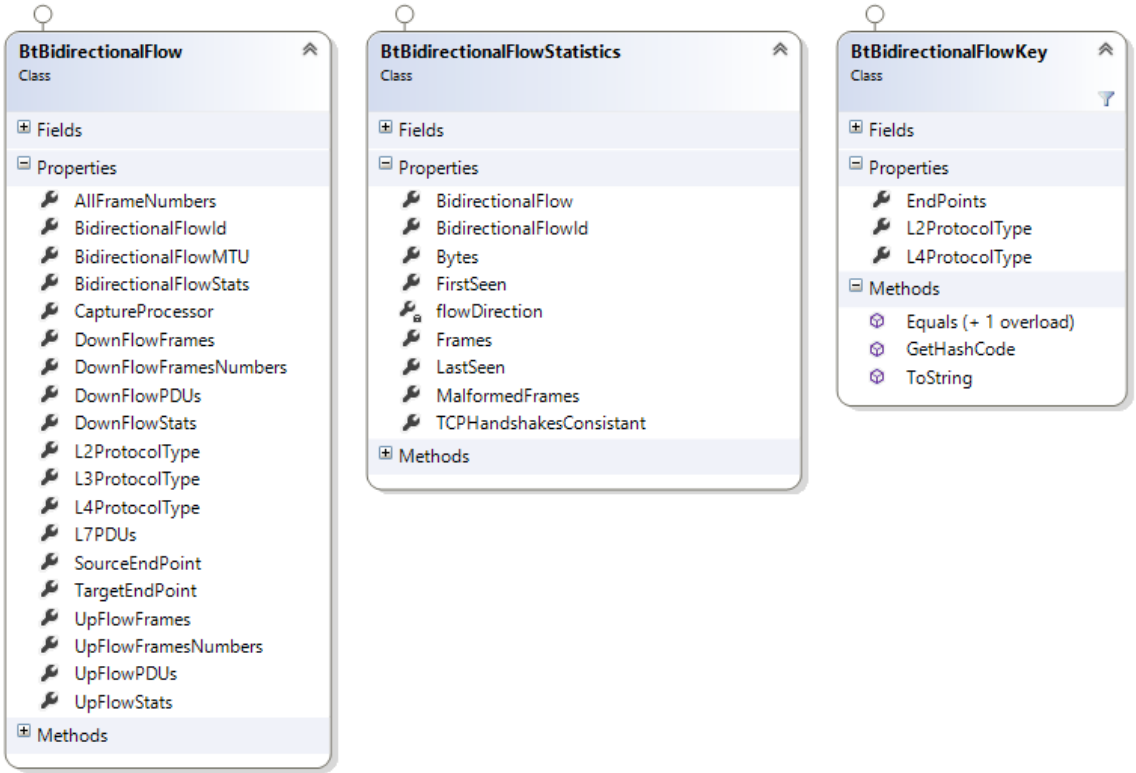


Figure 4.12: The class diagram describing inner members of *BidirectionalFlow*, *BidirectionalFlowStatistics* and *BidirectionalFlowKey* classes.

(Source|Destination)EndPoint, etc. are filled during the same pass as the flow tracking. Other properties providing the aggregated overview like *(Up|Down)FlowPDUs*, *(Up|Down)FlowStats* are evaluated on demand, therefore, computing resources are not wasted on their precomputation.

At this moment, flows are stored in a dictionary that provides the fastest access to the appropriate flow record corresponding to the given key. In reference to a future planing expansion to support big data processing, the bidirectional flow tracking mechanism is design to be easily reimplemented using inheritance from *BidirectionalFlowTrackerBase* class to fulfill newly occurred requirements.

On demand, at any time after the bidirectional flow tracking is finished, the *BidirectionalFlow* can provide statistics that could be easily expanded in future and together with the collection of tracked bidirectional flows are meant to provide the briefest overview about captured communication.

UnidirectionalFlow

The *UnidirectionalFlows* are meant to hold communication in one direction as a part of the separated transport protocol session from the *BidirectionalFlow*. The separation of the bidirectional flow to transport protocol sessions is done by the *L4Processor* which, according to used transport protocol and various heuristics, is trying to determine related parts of communication and group them as a unidirectional flows.

On the Figure 4.13 is presented a structure of the *UnidirectionalFlow*. The class holds meta-information extracted from the flow to be used to pair two unidirectional flows to one conversation representing the transport layer protocol session.

The *UnidirectionalFlow* is also used as a storage for collected application protocol PDUs by the *L4Processor* and encapsulates creation and manipulation with them. This centralized control over the *L7PDUs* is useful for debugging and testing purposes, because several constraining conditions this controller could be applied .

The *UnidirectionalFlows* stores the references to belonging frames. There are three categories of these frames.

- Frames that *carries data* are stored in corresponding *L7PDU* objects.
- Frames that are *filling space* after frames that are missing from the capture file.
- Frames that do *not carry any data* but are used for signalization for transport layer protocol logic.

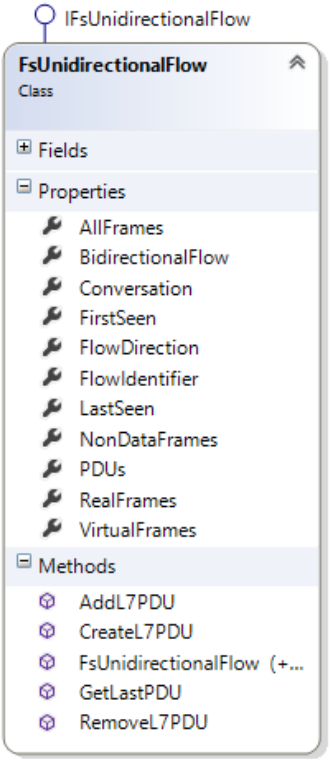


Figure 4.13: Class diagram describing inner members of UnidirectionalFlow class.

It would be counterproductive to store all belonging frames frames in the *UnidirectionalFlow* object because it would be redundant and a space wasting. Because of that it was decided to store frames at the most specific segmentation level that is the PDU object. Objects higher in the hierarchy are then aggregating the information stored in the lower objects and providing it on demand.

The *UnidirectionalFlow* provides an interface to access basic properties of the flow. Basic attributes are first, last frame's time stamp and a flow identifier. The *FlowIdentifier* is a sequence number of *SYN* packet or in the other direction the acknowledgement number of *SYN*, *ACK* packet when TCP transport protocol is used. The flow identifier could be any other flow property that could distinguish transport protocol sessions from each other and can be obtained from frames in each flow separately.

FlowStore

The *FlowStore* is similar to other storing components like *BidirectionalFlowTrackerInMemory* (see section 4.2.2) and the *ConversationStore* (see section 4.2.2) is used to encapsulate the functionality as a collector of objects (in this case unidirectional flows), adds an additional business logic and is prepared for future expansion to the cloud computing environment.

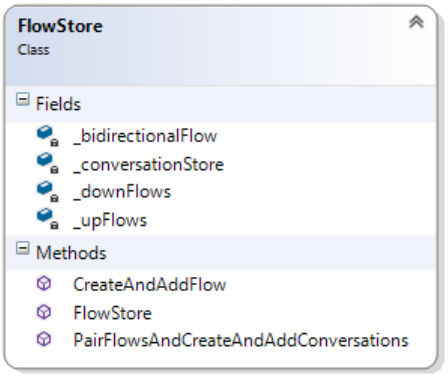


Figure 4.14: Class diagram describing inner members of FlowStore class.

The additional business logic implemented in the Flow store takes care of pairing unidirectional flows to the conversations. The pairing mechanism might not be always correct, because it is using heuristics to find the best matching flows.

The flow pairing is based primary on a recognition of TCP handshake and uses *Sequence* and *Acknowledgement* numbers of first two packets to pair both unidirectional flows and creates an conversation of them.

In case, that the communication is not captured completely, is damaged and/or transport protocol is UDP, the algorithm tries to find the best matching flows by their first timestamps. If flows still cannot be paired, since there are too much of them with similar characteristics, new conversations will be created each for a single unidirectional flow with missing flow in other direction.

ConversationValue and ConversationStore

The *ConversationValue* is the most important structure in the whole *Netfox.Framework* . It is the identifier of the smallest unit that could be used as a data source for the reconstruction process. The *ConversationValue* also holds the meta information about the conversation, ergo. the transport protocol session. For detailed enumeration of these information see Figure 4.15.

Each conversation value has a reference to the parent *BidirectionalFlow* to provide a fast mechanism to find another conversations in case that the *BidirectionalFlow* was wrongly separated to the conversation and some additional data are needed.

The conversation it self can be identified in two ways. Firstly, using the reference to the *ConversationValue* object is the simplest way, all meta information is provided in the *ConversationValue* object, but the reference is not persistent. Therefore, there is the second way, each *ConversationValue* object has its unique identifier the *ConversationId*, which is mainly used in cooperation with the *FrameworkController* module.

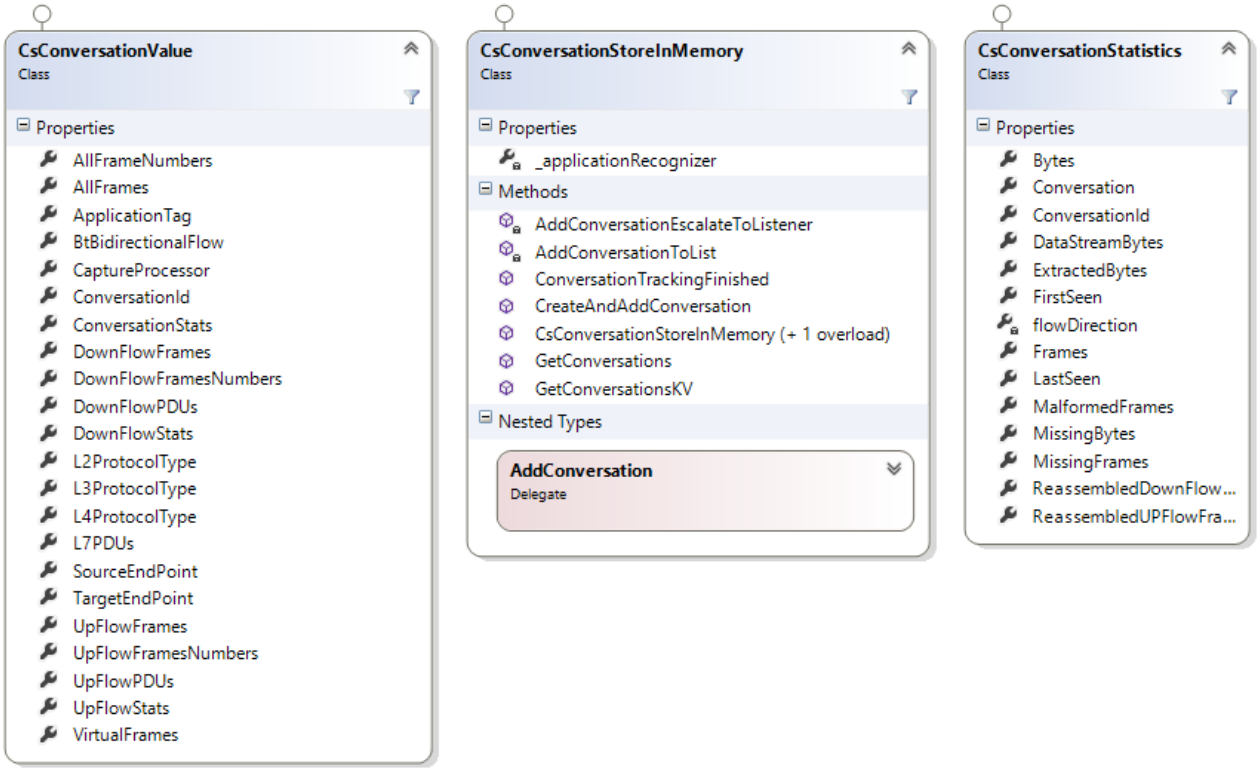


Figure 4.15: Class diagrams describing inner members of ConversationStore classes.

Nearly the same importance as the *ConversationValue* holds the *ConversationStatistics* object. It is required to provide information about the quality of the reconstructed data even before the application protocol reconstruction it self. Therefore, *ConversationStatistics* provides information gathered from a result of the TCP reassembling process concerning extracted and/or missing data size, malformed and/or missing frames and a whole conversation size.

The *ConversationStore* is another of storing components with the purpose of adding an additional business logic to the process of collecting *ConversationValue* objects. *Conversations* are, unlike the *BidirectionalFlows* complete at the time of their storage, therefore, they could be immediately escalated to registered event handlers for imminent processing. This is very efficient mechanism, because conversation separation is a part of reassembling process, which is not inconsiderable time consuming predominantly on a big data.

After all conversations are separated and stored in the *ConversationStore*, the *ConversationTrackingFinised* method is called to transform inner conversations storage into structures with duplicate information, but different item accessing properties. The first one is a *Dictionary* mapping the *ConversationId* on the *ConversationValue* reference providing the best possible performance in accessing the item by its key with access complexity of O(1). The second one is an array of *ConversationValues* which provides the best possible performance for operations with enumerable characteristics and is a thread safe.

L4Processor

The *L4Processor* is the hearth of the reconstruction framework. This module is responsible for the *separation of conversations*, *IP defragmentation*, *TCP reassembling* and *application layer PDU creation*. The frame processing is carried out in several ways according to the frames's transport protocol. Current implementation is supporting UDP and TCP transport layer protocols. If the need to process larger variety of protocols arise, the framework is prepare to support it.

The purpose of this module is dual. The first is a separation of frames stored in a bidirectional flow to the transport protocol sessions. One bidirectional flow can contain several transport protocol sessions which can be continued in sequence or even overlap one another. This module implement mechanisms that can detect this behaviour and separate sessions. The second is the transport protocol processing and preparation of L7PDUs (see Figure 4.16) by the properties of that protocol. The L7PDUs are objects caring ordered list of frames obtained by the reassembling process in case that transport protocol was the TCP. If the UDP were used, for each UDP frame payload one L7PDU would be created.

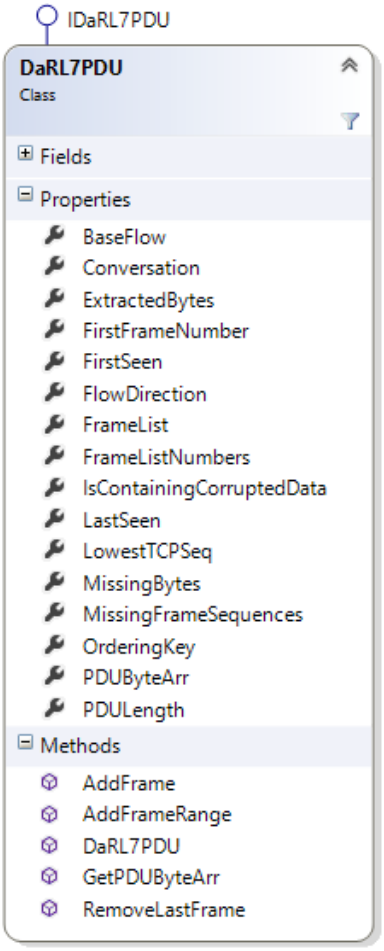


Figure 4.16: Class diagram describing inner members of *L7PDU* class.

UDP conversation separation

The conversation separation based on UDP sessions is a relatively easy operation that has only one parameter that is maximal time span between two transmitted or received frames. Because the UDP protocol has no mechanism of packet identification other then the time span, there is no way to separate two concurrent UDP sessions mixed up together by a capture file merging. Separation of that sessions would only be possible with a knowledge of an application protocol they carry and therefore they are left for application protocol parser to deal with. The separation using time span is definitely not a clean solution, but it is relatively the best there is.

The time span interval should be subjected to a future research to find the best delimiting value or an algorithm to analyze the actual flow and dynamically change the time span limit using some heuristic to separate UDP sessions. Possible heuristic might be a continued calculation weighted arithmetic mean of time stamps differences with empirically estimated weights.

TCP conversation separation and reassembling

The conversation separation based on TCP sessions is a non trivial operation that is using various heuristics to separate TCP sessions in a given bidirectional flow. Possible combinations of multiple TCP session was announced on subsection 4.2.1. Implemented conversation separation algorithm should consider all mentioned combinations of merged flows and try to separate them using an available information provided in a TCP header combined with an empirically gained knowledge obtained by observation and analysis of samples of captured communication.

The TCP provides various information (see Figure 4.2) that could be used in session reconstruction. The finite state machine enlightening how the TCP works is presented on Figure 4.3. Necessary properties extracted from the TCP header used in current implementation are *sequence* and *acknowledgement* numbers, *FIN*, *SYN*, *ACK*, *RST* flags, *check-sum* and *window size*.

The first step of reassembling is a separate analysis of flows stored in the *BidirectionalFlow*. Analysis is trying to find packets that are participating on a *TCP handshake*. Found packets are ordered by their *windows size*, because reassembling stream with a smaller *TCP windows* gives more accurate results. Those packets are used in a row as a initiation point where the *TCP reassembling* starts. The *sequence* and the *acknowledgement* numbers of those packets are used in a algorithm that is pairing both separately reassembled unidirectional flows.

For a faster iteration through a frame collection a DaRFrameCollection was implemented. The DaRFrameCollection keeps all frames ordered by their sequence numbers and because it is build on an LinkedIterableList, the addition and/or removal operation after and/or before item with known reference have an constant complexity. That is very efficient, because the collection is always processed in enumeration and evaluated frames are removed, therefore, there is a N removal operation on a collection of size N. The DaRFrameCollection override a GetEnumerator method, substituting it with a special implementation of enumerator that can recover from the base collection change in the time of enumeration.

The conversation separation and reassembling process begins with the initiation frame (participating in TCP handshake) with the smallest *TCP window* and continues for each another in a row. When all initiation frames have been used and the *DaRFrameCollection* for that flow is not empty, the frame with the smallest *sequence* number is used as the initiation frame, but resulting conversation will contains only one direction of communication, because without the *TCP handshake*, the pairing algorithm would not match the unidirectional flows for now.

The *reassembling algorithm* it self proceeds as follow. The enumerator is created and and enumerate it self until the the initial frame is found. The flow identifier is created according the frame role in *TCP handshake* (see Listing 4.2 lines 20-23). The next expected *TCP sequence number* is derived by the current one (see Listing 4.2 lines 24-28). Now, when initiation is done, reassembling process can enumerate over the all frames ordered by *sequence numbers* in a *current flow direction*. Every frame is checked if is not malformed up to transport protocol layer (Listing 4.2 line 31)). In this step the *TCP checksum* might be verified as well or the error might be detected by the *TCP retransmission*. After frame verification, there is a place for heuristics determining the frame belonging to the *TCP session* application (time based heuristic Listing 4.2 lines 33-34)).

When it is certain that the frame is a part of current *TCP session*, the frame‘s transport layer payload position in a data stream is computed by the deduction of expected and current *TCP sequence numbers*. There could be three cases. The deduction is positive means that there are some missing data that was not captured and missing space has to be marked and filled by the virtual frames. The negative deduction signalized that the current *sequence number* is lesser then expected and the retransmission of last frame might occurred or the *TCP keep alive* datagram was send. In case of retransmission, the current frame‘s *TCP checksum* is verified and if it is correct the last frame from the current PDU is exchanged for the current frame.

When the *sequence numbers* deduction is *equal to zero* as it should be in a majority of cases, it is a signal that no data are missing and the current frame is a continuation of the last one. In a case that the *L7PayloadLength* is *greated then zero*, the next expected frame number is calculated and current frame is added to the current PDU‘s frame list. In rare cases, when the packet is IPv4 fragmented, it cannot be added only by it self, but all fragments must be defragmented and added as a sequence (see Listing 4.3 lines 28-24). All frames has to be checked whether they have set *FIN* or *RST* flags (see Listing 4.3 lines 37-38). If they do, it is a signal that the current TCP session is at the end and a communicating side has closed the connection. If they do not, the settings of TCP *PHS* flag is checked and/or if a length of current frame‘s *L7PayloadLength* is *lesser then MTU* then it is a sign that application message is at the end and a new *L7PDU* is created and added to the current unidirectional flow.

```
public void RunL4TCPProcessor(){
  //DaRFrameCollection implementation of double linked list with predefined
  //frame ordering relation by TCP Seq number
  _upFlowFrames = new DaRFrameCollection(bidirectionalFlow.UpFlowFrames);
  _downFlowFrames =new DaRFrameCollection(bidirectionalFlow.DownFlowFrames);
  //TCPHandshakeEvaluator analyses both directions of flow and tries to find
  //all TCP handshakes. SYN and SYN+ACK packets from each flow are grouped
  //and ordered by Seq number
  tcpHandEval = new
    TCPHandshakeEvaluator(bidirectionalFlow,_upFlowFrames,_downFlowFrames);
  foreach (var initFrame in tcpHandEval.UpInitFrames + _upFlowFrames)
    ProcessTcpSession(initFrame);
  foreach (var initFrame in tcpHandEval.DownInitFrames + _downFlowFrames)
    ProcessTcpSession(initFrame);
}

private void ProcessTcpSession(IPmFrame synFrame, DaRFrameCollection flow){
  _flowEnumerator = flow.GetEnumerator(synFrame);
  _lastTimestamp = _currentFrame.TimeStamp;
  if (_currentFrame.isSynPacket) //SYN packet
    _currentFlow.FlowIdentifier = _currentFrame.TcpSequenceNumber+1;
  else if (_currentFrame.isSynAck) //SYN+ACK packet
    _currentFlow.FlowIdentifier = _currentFrame.TcpAcknowledgementNumber;
  if (_currentFrame.TcpFSyn){
    _expectedSequenceNumber = _currentFrame.TcpSequenceNumber + 1;
    NextFrame(); }
  else
    _expectedSequenceNumber = _currentFrame.TcpSequenceNumber;
  while (_currentFrame != null &&!_currentFrame.TcpFSyn){
    //Skipping malformed frame from data, storing it in ono data frames
    if(_currentFrame.IsMalformed) {AddNonDataFrame();NextFrame();continue;}
    //Skipping frames that do not fit in to time windows
    if (Math.Abs(_currentFrame.TimeStamp - _lastTimestamp.TotalSeconds)
        > TCP_SESSION_ALIVE_TIMEOUT) {SkipFrame();continue;}
    //Difference in Seq number deduction with expected other then 0 may
    //signify missing frames or retransmits
    long x = _currentFrame.TcpSequenceNumber - _expectedSequenceNumber;
    //Missing TCP segment, fill it with virtual frame
    if (x > 0) {TCPMissingSegment(x);}
    //TCP segment is retransmitted or overlapped
    else if (x < 0) { TCPRetransmit();}
    //else if (x == 0) //Normal TCP packet -- next in the sequence
    TCPNormalSequence();
    NextFrame(); //Move to next frame in row
  }
}
```

Listing 4.2: Reassembling algorithm in pseudocode.

```
private void TCPMissingSegment(long x){
  //Create a virtual frame for stuffing missing space
  IPmFrame virtualFrame =
    CaptureProcessor.Capture.AddVirtualFrame((uint) x,
                                _currentFrame.TimeStamp.AddTicks(-1));
  _currentFlow.VirtualFrames.Add(virtualFrame);
  _currentPDU.AddFrame(virtualFrame);
}

private void TCPRetransmit(){
  //Keep alive do not increment Seq, might be mistaken fro retransmits
  if(_currentFrame.isTCPKeepAlive)
    {AddNonDataFrame();NextFrame();continue;}
  //Retransmited frame, exchange it with last in _currentPDU
  //The last one was probably damaged
  if(_currentFrame.IsValidChecksum){
    if(_currentFrame.TcpFFin || _currentFrame.TcpFRst)
      {AddNonDataFrame();NextFrame();break;}
    CheckOverlepAndExchangeWithLastPDUFrame(_currentFrame); continue;
  }
}
private void TCPNormalSequence(){
  //Data segment to store
  if (_currentFrame.L7PayloadLength > 0){
    _expectedSequenceNumber =
      _currentFrame.TcpSequenceNumber + _currentFrame.L7PayloadLength;
    //IPv4 fragmented packet
        if (_currentFrame.Ipv4FMf){
          var defragmentedFrames = FindFragments(_currentFrame);
          _currentPDU.AddFrameRange(defragmentedFrames);
        }
        //Normal state, no IPv4 fragmentation
        else
          _currentPDU.AddFrame(_currentFrame);
  }
  //TcpFFin or TcpFRst signifies the end of current conversation
  if (_currentFrame.TcpFFin || _currentFrame.TcpFRst)
    {AddNonDataFrame();NextFrame();break;}
  //TcpFPsh or L7PayloadLength<MTU probably signifies an end of message
  if ((_currentFrame.TcpFPsh ||
       _currentFrame.L7PayloadLength <
       _bidirectionalFlow.BidirectionalFlowMTU)
       && _currentPDU.FrameList.Any())
    {_currentPDU = null;} //null assignment forces a creation of new PDU
}
```

Listing 4.3: Reassembling algorithm in pseudocode.

Chapter 5

Benchmark and comparison with existing tools

In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it. The term 'benchmark' is also mostly utilized for the purposes of elaborately-designed benchmarking programs themselves. Benchmarks provide a method of comparing the performance of various subsystems across different chip/system architectures. (Wikipedia 2014a).

The benchmarking method was chosen to provide a general picture about a quality of the implemented solution. In processing pipeline there are several checkpoints that can be considered as points of synchronization (serialization), where all operations evaluated sequentially and/or in parallel have to be finished to proceed to another section. Considering a unique design of *Netfox.Framework*, only lower level components were chosen for the benchmarking, because their functionality can be compared to other existing tools, at least at some level of complexity.

For more information about the *performance of higher level processing*, especialy the data reconstruction and their exportation done by the Netfox.Framework and a visualization application a *Netfox.Detective* (see Mareš 2014).

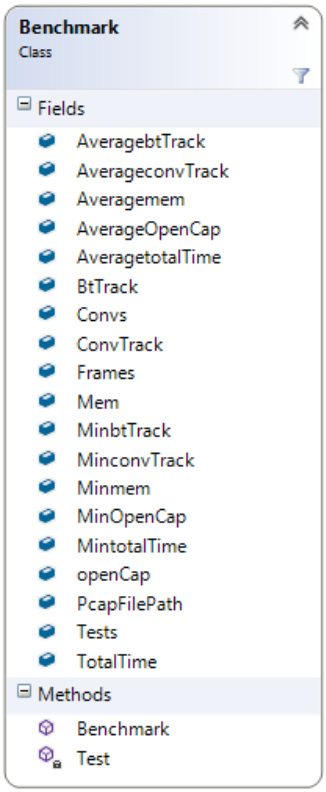


Figure 5.1: The class diagram describing inner members of Benchmark class with all performance counters.

According to the Figure 4.9, for the measurement of a *user time* these sections (CaptureProcessor's methods) were chosen: *capture file opening*, *bidirectional flow tracking*, *conversation tracking* and the *total time* of all of them. The capture file opening time depends on an existence of *capture file index*. If the *capture file index* exists, required information is taken from it, resulting in faster evaluation, because the actual capture file is not parsed again. Index contains several offsets pointing to the begin of L2, L3, L4 headers of every frame in the capture file. Therefore, needed information is retrieved

in a constant time. When the capture file is processed for the first time, the index file is created and saved for a future usage.

The second measured variable is a *consumed memory* which is significantly impacted by a size of captured file. Many existing tools allocates even more memory then is a size of capture file. This behavior is unreasonable and that software may not be able to process large capture files without specialize hardware. The consumed memory does not so much depend on the file size as it may seem, but on the number of frames inside the capture file. For every frame there is allocated one or more structures carrying frame‘s meta-information.

To ensure the most accurate results as a primary benchmark testing set, capture file sets were generated containing *two* and/or *four conversations* each in various sizes of *12, 25, 50, 75, 100, 150* and *200MB* (from now on might be also referred to as a *generated traffic*). Each frame contains only one byte of payload data to minimize the size of the capture file dismissing unused data and ensuring that other tools would not apply application layer dissectors which are not applied in the benchmark of *Netfox.Framework* as well. Assuming that the most common IP packet size (including the IP header) with distribution of 58.333333% in packets of normal traffic is 40B (ergo 54B including encapsulation in Ethernet) according to John and Tafvelin 2007. Our generated traffic with the size of 55B is near the most common value and ensures application of TCP reassembling process in the *Netfox.Framework*. On the other hand, the most common packet size with distribution of 56% in bytes is 576B, therefore, it might be assumed that measured statistics of these data sets would correlate with a $14 - 15$ times greater capture files of normal traffic.

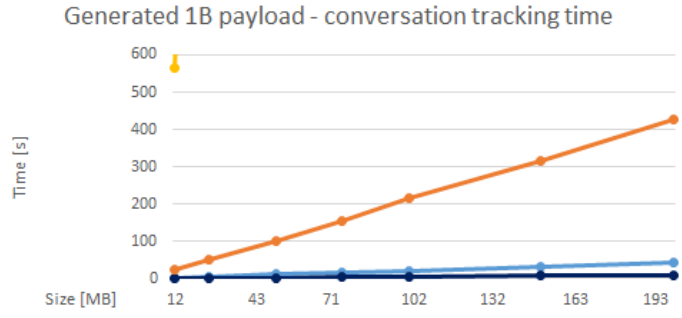
The benchmark test was created as a part of *UnitTests* and is available to use with the *Netfox.Framework* to ensure that there are not any performance problems on given hardware architecture. For all available inner members (performance counters) see Figure 5.1. The *BenchmarkTest* runs *N times* (by default 10 times, at least 2) capture file processing through *PmLib‘s file opening*, *bidirectional flow tracking* and *conversation tracking*. Before the first run, the benchmark deletes the capture file index, therefore in the first iteration capture file parsing will be included as well to provide results comparable with other tools. Other iterations are measuring the performance of processing pipeline components themselves and simulating second and following time of opening. After the end of benchmarking, final statistics (average and minimal value) are computed from second and next results.

All stated performance measuring was committed on identical hardware with specification as proceed.

- *CPU* – Mobile DualCore Intel Core i5-3380M, 3600 MHz (36 x 100)
  - CPU Alias Ivy Bridge-MB
  - Min/Max Multiplier 12x/29x
  - L1 Code Cache 32KB per core
  - L2 Cache 256KB per core (On-Die, ECC, Full-Speed)
  - L3 Cache 3MB (On-Die, ECC, Full-Speed)
- *Memory* – 8GB Dual DDR3 SDRAM
  - Bus Width 128-bit
  - Real Clock 667 MHz (DDR)
  - Effective Clock 1333 MHz

- *Hard drive* – Samsung SSD 840 EVO 250GB (232 GB)
- *Operating system* – Microsoft Windows 8.1 Professional
  - Kernel Type Multiprocessor Free (64-bit)
  - OS Version 6.3.9600.17041 (Win8.1 RTM)
  - .NET Framework 4.0.30319.33440 built by: FX45W81RTMREL

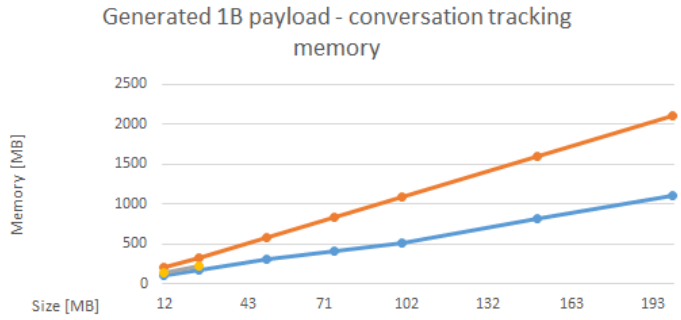
For the comparison with other existing tool, widely used in *Law enforcement agencies*LEAs, the first generated capture set containing two conversations in various capture file sizes was selected. The *time consumption* of capture processing in *Wireshark* and *Network monitor* was measured manually using stopwatch, on the other hand the *Netfox.Framework* was measured using *BenchmarkTest*. Measurement results were entered into the table and plotted into the linear chart, see Figure 5.2. Results had shown that in evaluation of this type of data the *Netfox.Framework* outperformed competitive tools and is several times faster then the *Network Monitor* and even more times then *Wireshark* which performance was measured only for 12 and 25MB capture files, because its performance with this type of capture files was too low.



	12	25	50	75	100	150	200
Conversations	2	2	2	2	2	2	2
Frames	145448	290903	581812	872721	1159994	1739994	2329994
NFX 1st opening	2.823	5.851	11.492	17.078	20.850	31.337	44.934
NFX 2nd opening	0.561	1.230	2.587	3.652	4.752	7.383	9.880
NetworkMonitor	24.77	51.26	101.32	152.61	215.23	315.35	428.78
Wireshark	563.4	2518.23					

Figure 5.2: The chart comparing a performance *Netfox.Framework*, *Wireshark* and *Microsoft Network Monitor* in processing of generated capture files with only 1B payload.

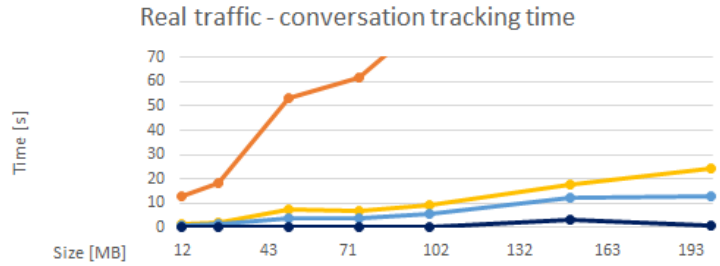
Another measured aspect, except the time of processing, is a required memory. As mentioned above, application usually stores every frame (at least meta information about it) in a memory. Inner members of structures are almost always aligned, because of performance reasons, according to given architecture, the padding resulted by alignment might be in many cases terrible waste of memory. The Figure 5.3 shows comparison of memory consumption of each tested application. The *Netfox.Framework* came out with the best, ergo the lowest memory consumption, but it is still almost six times greater then the capture file. The *Wireshark* seems to be only 1.27 times more demanding, but *Network Monitor* almost 1.9 times. On charts it is also obvious that both Netfox.Framework and Network monitor have a linear complexity in the time and memory.



	12	25	50	75	100	150	200
NFX	110.166	178.168	311.484	402.010	515.244	817.476	1,101.246
NetworkMonitor	203	330	582	835	1,085	1,589	2,101
Wireshark	140	229					

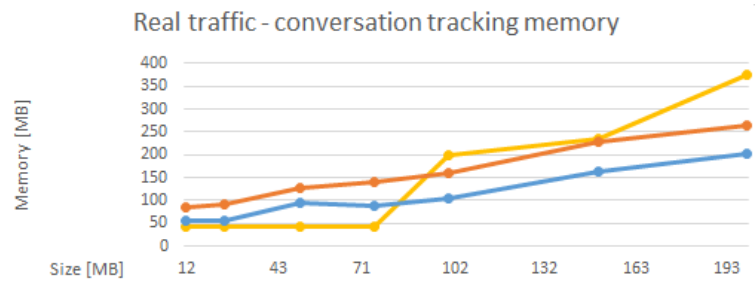
Figure 5.3: The chart comparing a required memory of *Netfox.Framework*, *Wireshark* and *Microsoft Network Monitor* in processing of generated capture files with only 1B payload.

As a second test set, a real traffic was used, captured on the access switch in the laboratory where students were asked to surf the internet and commit their normal behavior on a network. The resulting capture file was afterwards split into pieces that were correlating with sizes of generated set. The distribution of conversations in those file is not a linear function of their sizes and neither is the distribution of frames. On the other hand, the resulting approximated time function shown on Figure 5.4 exhibiting a linear characteristics. The same behavior could be seen on Figure 5.5 describing memory. On the grounds of the performance comparison it is evident that *Wireshark* is optimized on real like traffic characteristic, because it is showing a great performance improvement compared to the generated traffic, and is only 1.5-2 times slower then the *Netfox.Framework*.



	12	25	50	75	100	150	200
Conversations	358	8	1859	1179	624	5211	5461
Frames	10011	16954	48796	55204	85064	183251	173348
Wireshark	1.59	1.96	7.45	7.11	9.42	17.75	24.13
Network Monitor	12.95	18.22	53.28	61.72	86.25	153.18	164.45
NFX 1st opening	0.964	1.255	3.691	3.875	5.449	12.425	12.931
NFX 2nd opening	0.014	0.016	0.078	0.077	0.155	3.483	0.542

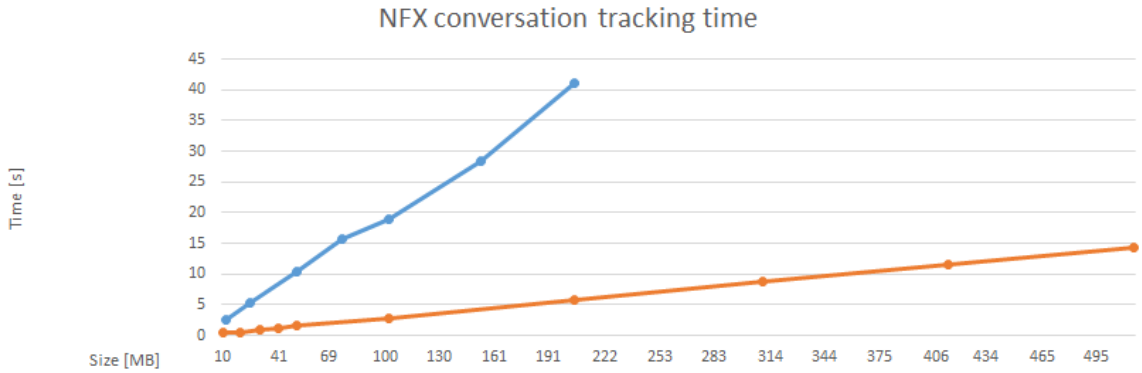
Figure 5.4: The chart comparing a performance of *Netfox.Framework*, *Wireshark* and *Microsoft Network Monitor* in processing of capture files with a real traffic.



	12	25	50	75	100	150	200
Wireshark	42.40	42.40	42.40	42.40	199.60	236.10	373.50
Network Monitor	85.80	92.10	127.30	140.80	159.90	228.90	265.10
NFX	53.957	54.227	95.511	86.606	104.980	162.361	201.228

Figure 5.5: The chart comparing a required memory of *Netfox.Framework*, *Wireshark* and *Microsoft Network Monitor* in processing of capture files with a real traffic.

To support the claim that the time needed to process the capture file is not a function of its size, but more likely its on a count of captured frames and slightly less on a conversations involved is presented Figure 5.6. Apparently, for a generated traffic where the size of capture file and number of frames are in a direct correlation, the resulting function is again almost linear as well as for the real traffic where this property is almost the same. The difference between both traffic sets is in a average size of frame, therefore, the number of frames in files of the same size for each traffic set. Therefore, the generated traffic has a larger slope of the line compared to real one.



	10	12	20	25	31	41	51	76	102	153	205	310	413	516
Real traffic frames	8239		16487		24738	32973	41224		82544		165316	248284	331342	413912
Real traffic	0.487		0.585		0.930	1.149	1.675		2.891		5.726	8.695	11.621	14.392
Generated traffic frames		145448		290903			581812	872721	1159994	1739994	2329994			
Generated 1B payload		2.580		5.333			10.514	15.667	18.865	28.300	41.125			

Figure 5.6: The chart comparing a performance *Netfox.Framework* in processing of capture files with a real and generated traffic.

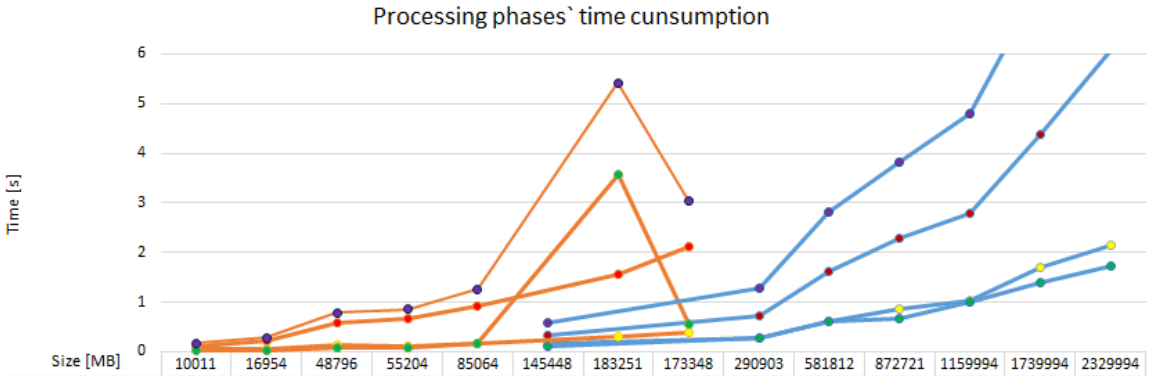
Based on a previous observation, the last chart Figure 5.7 is showing individual phases of processing pipeline related to the number of frames in each capture file. On a first sight, there is apparent huge loss of performance in processing the real traffic capture file of size 25MB. This lost is caused by the retransmitted frames, that have to be whole retrieved from the capture file, parsed and their consistency has to be checked to decide which one is going to be added to the list of reassembled (reordered, filtered) frames.

As it is evident on the given chart that traffic samples of a real traffic with comparable number of frames to the generated traffic took always more time to process, because there are much more objects representing frames to be created and their allocation and initialization take significant amount of time.

Based on results gain by the previous measurement could be observed several rules of dependence to estimate an amount of time needed for a capture file processing.

- *capture file size* – inevitable variable, data of that size have to be read from a storage to be processed, the estimated speed depends on the type and reading speed of the storage
- *number of frames* – more accurate indicator, because for one frame there is an object representing it, but nevertheless that objects has to be initialized by equal size of data: *average size of frame* $*$ *number of frames* $\approx$ *capture file size*
- *number of bidirectional flows* – this number determines a degree of potential parallelism during conversation tracking
- *number of conversations* – *number of conversations* $\gg$ then *number of bidirectional flows* signifies majority of TCP sessions between two hosts and most probably the same service
- *number of TCP retransmitted frames* – this number significantly affects the time of evaluation, because every retransmitted frame has to be verified by check-sum calculation

More detailed statistics are not provided because of their complexity and extent. To get these details it is advice to use the *Netfox.Detective* which uses *Netfox.Framework* to obtain them. The *Netfox.Detective* with all of testing capture files is presented on enclosed DVD.



Size [MB]	10011	16954	48796	55204	85064	145448	183251	173348	290903	581812	872721	1159994	1739994	2329994
Convs	358	8	1859	1179	624	2	5211	5461	2	2	2	2	2	2
PCAP size	12	25	50	75	100	12	25	50	25	50	75	100	150	200
Gen open						0.323			0.733	1.603	2.289	2.783	4.368	6.089
Real open	0.119	0.220	0.566	0.661	0.927		1.545	2.103						
Gen bt track						0.152			0.285	0.605	0.854	1.021	1.691	2.142
Real bt track	0.040	0.045	0.138	0.114	0.169		0.302	0.393						
Gen conv track						0.101			0.264	0.608	0.676	0.990	1.377	1.713
Real conv track	0.015	0.018	0.080	0.078	0.164		3.568	0.554						
Gen total						0.576			1.282	2.816	3.819	4.794	7.436	9.944
Real total	0.174	0.283	0.784	0.853	1.260		5.415	3.050						

Figure 5.7: The chart comparing a performance of individual phases of *Netfox.Framework* in processing of capture files with a real and generated traffic.

Chapter 6

Conclusion

The *Network forensic analysis* is an increasingly discussed topic in the recent decade because of a rapidly raising number of criminal activities employing a network infrastructure. As computer networks grow and a new equipment is being connected every second, a crucial need for an efficient network monitoring tool arises. Two basic methods are applied. Firstly, collecting traffic metadata in a form of Netflow records, which are often applied in solutions of data retention, to provide an evidence of intercommunication of network devices. Secondly, full communication capturing followed by the subsequent detailed analysis is applied in specific cases, when a target of an investigation is known.

We present a network forensic platform, called *Netfox.Framework* (NFX), which has been developed as an open-source, extensible, and modular analytic software framework, providing a conversation-based approach usable for an advanced data-mining in a captured communication. The NFX development is driven by the need of providing a robust method to reduce a complexity and a time during a development of various specific network forensic applications. Almost every possible forensic investigation use-case requires to reconstruct, at least partially, an application data layer. The functionality implemented in the NFX resembles not only the implementation of TCP/IP stack at end nodes, but also other mechanisms necessary to understand bidirectional communication up to application protocol layers.

The thesis discusses properties of the *Netfox.Framework*, its overall architecture and information on selected areas of its design and implementation. The performance of the NFX implementation was evaluated on several available data sets. Based on the evaluation, we claim that the NFX can be readily employed for the network forensic tool implementation. To support this, it was implemented by *Bc. Marting Mareš* an experimental network forensic tool with the advanced GUI testing capabilities of the NFX. The current version of the NFX represents a fully working proof of concept supporting common network and transport protocols, and providing a data-mining functionality for selected application protocols, namely, *HTTP*, *IMAP*, *SMTP*, *POP3*, *OSCAR*, *XMPP*, *YMSG*, and *MSN*. The future development of the NFX is focused on providing a scalable solution that can be deployed in a distributed environment to handle big forensic data. Also, the current research is aiming on an intelligent application data classification and processing. These intelligent methods will be implemented as plugins to the framework performing specific analytic functions. Based on preliminary experiments with a real world data, we also identified the need of robust methods for processing lower-layer protocols including the data decapsulation from a tunneled traffic.

The Netfox.Framework have been implemented using *C#* programming language and a *.NET 4.5 framework*. The majority of code was covered by UnitTests to prove a validity of implementation. UnitTests by themselves are also used as more or less complex examples of a usage of individual components and/or more complicated use-cases. Every low level component like BidirectionalFlowTracker, ConversationTracker, ApplicationRecognizer have tested every method and line of code, because its output is well known thanks to the low level of their abstraction. This test would identify an error precisely if any might occur in a future functionality extension. Components that lies in the middle of an abstraction providing a control function of lower components are tested based on output of lower ones. Their tests provides automatic validation of basic functionality during a change-set commit. These tests are fast but would not identify exact point where the bug might occur. Components on the top level of abstraction, meaning application protocol dissectors and evaluators (Sleuths) are not tested line by line, but there are some statistics calculated by using data they produce. Possible differences in the reference statistic and newly calculated one would indicate a change or an error in lower components, therefore further investigation is required.

The Netfox project is being developed in collaboration *Czech law enforcement agencies* (LEAs) and is aiming to be a useful tool set helping with an investigation of a cybercriminality. The current implementation is a fully working proof of concept with a widely open space for an optimization, an extension and an everyday innovation. We aim at the part of network forensic investigation methods that focus on helping law enforcement units to investigate already known incidents and connected evidence gathering. Even though it is possible to use the Detective application in a wider perspective to look for known incident patterns using an advance querying system supporting predefined and/or user written queries. Based on real world experiences, the *Netfox.Detective* application have implemented several View-models to provide an investigator with the best user experience in his work and presents full reconstruction capabilities of *Netfox.Framework*. The tight cooperation in development of the Detective with the Framework provides a unique opportunity of testing implemented functionality in a real-like environment and give a fast feedback, instantly followed by bug fixes. Once we are satisfied with the optimization of framework modules based on a deeper profiling, and when computing resources of a single computer are insufficient, we are planning to port the *Netfox.Framework* into the cloud environment based on the Hadoop. That should bring new opportunities of evaluating big data captured in a long time period.

We are as well planning to ease demands on human resources needed to process reconstructed data by experimenting with a machine learning applied in the application protocol recognition, and an automatic network incident detection based on real investigation strategies.

Bibliography

Almulhem, A. and I. Traore (2005). *Experience with engineering a network forensics system.* In: Proceedings of the international conference on information networking (ICOIN 2005), Korea, LNCS 3391, pp. 62–71.

Berghel, H. (2003). *The discipline of Internet forensics.* Communications of the ACM 2003;46(8):15–20.

Broucek, V. and P. Turner (2001). *Forensic computing: developing a conceptual approach for an emerging academic discipline.* In: Fifth Australian Security Research Symposium;

Cohen, M. I. (2008). "PyFlag - An advanced network forensic framework". In: *Digital Investigation* 5.

Egevang, K. and P. Francis (1994). *RFC 1631: The IP Network Address Translator (NAT).* Status: INFORMATIONAL. URL: ftp://ftp.internic.net/rfc/rfc1631.txt, %20ftp://ftp.math.utah.edu/pub/rfc/rfc1631.txt.

Eoghan and Casey (2004). "Network traffic as a source of evidence: tool strengths". In: *weaknesses, and future needs, Digital Investigation* 1.1, pp. 28–43.

Garfinkel, Simson L. (2014). *TCP/IP packet demultiplexer.* https://github.com/simsong/tcpflow.

John, Wolfgang and Sven Tafvelin (2007). "Analysis of Internet Backbone Traffic and Header Anomalies Observed". In: *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement.* IMC '07. San Diego, California, USA: ACM, pp. 111–116. ISBN: 978-1-59593-908-1. DOI: 10.1145/1298306.1298321. URL: http://doi.acm.org/10.1145/1298306.1298321.

Mareš, Martin (2014). "Nástroj pro analýzu obsahu síťové komunikace". MA thesis. FIT BUT in Brno, Czech Republic.

Microsoft (2014). *NetworkMonitor blog.* URL: http://blogs.technet.com/b/netmon (visited on 01/03/2014).

Microtic (2010). *Manual:Connection oriented communication (TCP/IP).* URL: http://wiki.mikrotik.com/wiki/Manual:Connection_oriented_communication_(TCP/IP) (visited on 02/23/2014).

Palmer, G. (2001). *A road map for digital forensic research.* In: First digital forensic research workshop (DFRWS 2001) ; 2001, pp. 27–30.

Pilli, E. S., R. C. Joshi, and R. Niyogi (2010). "Network forensic frameworks: Survey and research challenges". In: *Digital Investigation* 7, pp. 1–2.

Ranum, M. (2013). *Network flight recorder.* (Visited on 12/26/2013).

Ren, W. and H. Jin (2005). *Modeling the network forensics behaviors.* In: Proceedings of the first international conference on security and privacy for emerging areas in communication networks (SecureComm 2005) ; Sept, pp. 1–8.

SEC6NET (2014). *Netfox Framework, Codeplex.* URL: https://netfoxframework.codeplex.com/ (visited on 01/06/2014).

Spurgeon, Charles E. (2000). *Ethernet: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly & Associates, Inc. ISBN: 1-56592-660-9.

Wikipedia (2012). *Packet.Net site*. URL: http://sourceforge.net/apps/mediawiki/packetnet/index.php?title=Main_Page (visited on 12/31/2013).

— (2014a). *Benchmark (computing) — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=Benchmark_(computing)&oldid=603318424 (visited on 05/09/2014).

— (2014b). *IP fragmentation — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=IP_fragmentation&oldid=596755398 (visited on 02/23/2014).

— (2014c). *IPv4 — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=IPv4&oldid=595917075 (visited on 02/23/2014).

— (2014d). *IPv6 — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=IPv6&oldid=596671983 (visited on 02/23/2014).

— (2014e). *Transmission Control Protocol — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=594981526 (visited on 02/23/2014).

— (2014f). *User Datagram Protocol — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=User_Datagram_Protocol&oldid=594953408 (visited on 02/23/2014).

Appendices

Appendix A

CD index

Demo_data\ContentBrowser
Demo_data\SleuthsManagerConsole
Demo_data\SleuthsManagerGUI
Demo_data\dotnetfx45_full_x86_x64.exe
Demo_data\README.pdf
Demo_data\Detective
Netfox_framework_source\ApplicationRecognizer
Netfox_framework_source\CaptureManager
Netfox_framework_source\ConversationTracker
Netfox_framework_source\Core
Netfox_framework_source\CoreController
Netfox_framework_source\CoreFactory
Netfox_framework_source\CoreMisc
Netfox_framework_source\DefragmentAndReassemble
Netfox_framework_source\Examples
Netfox_framework_source\Exporters
Netfox_framework_source\FlowExporter
Netfox_framework_source\FrameworkModels
Netfox_framework_source\FrameworkUnitTests
Netfox_framework_source\Helpers
Netfox_framework_source\MessageParser
Netfox_framework_source\NplangCompiler
Netfox_framework_source\PacketParser
Netfox_framework_source\PmLib
Netfox_framework_source\SimplifiedMessageParser
Netfox_framework_source\SleuthManagerGUI
Netfox_framework_source\SleuthsManagerConsole
Netfox_framework_source\UnitTests
Programmer_documentation\html_doc
Programmer_documentation\Netfox_Framework_doc.chm
Thesis_source_text

Appendix B

Description of Network Forensic Analysis Tools (NFATs)

This table describing existing NFATs was taken over from (Pilli, Joshi, and Niyogi 2010)

Table B.1: Description of Network Forensic Analysis Tools (NFATs).

Name of the NFAT	Description
NetIntercept	Captures network traffic and stores in pcap format, reassembles individual data streams, analyzes them by parsing to recognize the protocol, detects spoofing and generates a variety of reports from the results.
NetWitness	Captures all network traffic and reconstructs the network sessions to the application layer for automated alerting, monitoring, interactive analysis and review.
NetDetector	Captures intrusions, integrates signature-based anomaly detection, reconstructs application sessions and performs multi timescale analysis on diverse applications and protocols. It has an intuitive management console and full standards based reporting tools. It imports and exports data in a variety of formats.
Iris	Collects network traffic and reassembles it in its native session based format, reconstructs actual text of the session, replays traffic for audit trial of suspicious activity, provides a variety of statistical measurements and has advanced search and filtering mechanism for quick identification of data.
Infinistream	Utilizes intelligent Deep Packet Capture (iDPC) technology and performs real-time or back-in-time analysis. It does high-speed capture of rich packet details, statistical analysis of packet or flow based data and recognizes hundreds of applications. It uses sophisticated indexing and Smart Recording and Data Mining (SRDM) for optimization.
Solera DS 5150	DS 5150 is an appliance for high-speed data capture, complete indexed record of network traffic, filtering, regeneration and playback. DeepSee forensic suite has three softwares – Reports, Sonar and Search – to index, search and reconstruct all network traffic.

Continued on next page

Name of the NFAT	Description
OmniPeek	Provides real-time visibility into every part of the network. It has high capture capabilities, centralized console, distributed engines, and expert analysis. Omnipliance is a network recording appliance with a multi-terabyte disk farm and high-speed capture interfaces. OmniEngine software captures and stores network traffic. OmniPeek interface searches and mines captured data for specific information.
SilentRunner	Captures, analyzes and visualizes network activity by uncovering break-in attempts, abnormal usage, misuse and anomalies. It generates an interactive graphical representation of the series of events and correlates actual network traffic. It also plays back and reconstructs security incidents in their exact sequence.
NetworkMiner	Captures network traffic by live sniffing, performs host discovery, reassembles transferred files, identifies rogue hosts and assesses how much data leakage was affected by an attacker.
Xplico	Captures Internet traffic, dissects data at the protocol level, reconstructs and normalizes it for use in manipulators. The manipulators transcode, correlate and aggregate data for analysis and present the results in a visualized form.
PyFlag	An advanced forensic tool to analyze network captures in libpcap format while supporting a number of network protocols. It has the ability to recursively examine data at multiple levels and is ideally suited for network protocols which are typically layered. PyFlag parses the pcap files, extracts the packets and dissects them at low level protocols (IP, TCP or UDP). Related packets are collected into streams using reassembler. These streams are then dissected with higher level protocol dissectors (HTTP, IRC, etc) (Cohen, 2008).

Table B.1 – Description of Network Forensic Analysis Tools (NFATs).

Appendix C

Description of network security and monitoring (NSM) tools

This table describing existing network security and monitoring (NSM) tools was taken over from (E. S. Pilli and R. C. Joshi and R. Niyogi, 2010)Pilli, Joshi, and Niyogi 2010

Table C.1: Description of network security and monitoring (NSM) tools.

Name of the NSM tool	Description
TCPDump	A common packet sniffer and analyzer, runs in command line, intercepts and displays packets being transmitted over a network. It captures, displays, and stores all forms of network traffic in a variety of output formats. It will print packet data like timestamp, protocol, source and destination hosts and ports, flags, options, and sequence numbers.
Wireshark	Most popular network protocol analyzer. It can perform live capture in libpcap format, inspect and dissect hundreds of protocols, do offline analysis, and work on multiple platforms. It can read and write files in different file formats of other tools.
TCPFlow	Captures data transmitted as part of TCP connections (flows) and stores it for protocol analysis. It reconstructs actual data streams and stores in a separate file.TCPFlowunderstands sequence numbers and will correctly reconstruct data streams regardless of retransmissions or out-of-order delivery.
Flow-tools	Library to collect, send, process and generate reports fromNetFlowdata. Important tools in the suite are flow capture which collects and stores exported flows from a router, flow-cat concatenates flow files, flow report generates reports forNetFlowdata sets, and flow-filter filters flows based on export fields.
NfDump	A suite of tools working withNetFlowformat: nfcapd NetFlowcapture daemon reads theNetFlowdata from the network and stores it. NfDump – NetFlowdump reads theNetFlowdata from these files, displays them and creates statistics of flows, IP addresses, ports etc. nfprofile – NetFlowprofiler filters the NetFlowdata according to the specified filter sets and stores the filtered data. nfreplay – NetFlowreplay sends data over the network to another host.

Continued on next page

Name of the NFAT	Description
PADS	PADSis a portable, lightweight and intelligent network sniffer. It is a signature-based detection engine used to passively detect network assets. It can sniff TCP, ARP and ICMP traffic packets. It can move information about unique assets and services seen on the network into permanent storage, output it in CSV or MySQL format or present an user friendly report.
Argus	Processes packets in capture files are live data and generate detailed status reports of the 'flows' detected in the packet stream. The flow reports capture the semantics of every flow with a great deal of data reduction. The audit data are good for network forensics, non-repudiation, detecting very slow scans, and supporting zero-day events.
Nessus	Vulnerability scanner featuring high-speed discovery, configuration auditing, asset profiling, sensitive data discovery and vulnerability analysis.
Sebek	Kernel based data capture tool designed to capture all activity on a Honeypot. It records keystrokes of a session that is using encryption, recover files copied with SCP, capture passwords used to log in to remote system, and accomplish many other forensics related tasks.
TCPTrace	Produce different types of output containing information, such as elapsed time, bytes/segments which are sent and received, retransmissions, round trip times, window advertisements, and throughpu.
Ntop	Used for traffic measurement, network traffic monitoring, optimization, planning, and detection of network security violations. It provides support for both tracking ongoing attacks and identifying potential security holes including IP spoofing, network cards in promiscuous mode, denial of service attacks, trojan horses and port scan attacks.
TCPStat	Reports network interface statistics like bandwidth, number of packets, packets per second, average packet size, standard deviation of packet size and interface load by monitoring an interface or reading from libpcap file.
IOSNetFlow	Collects and measures IP packet attributes of each packet forwarded through routers or switches, groups similar packets into a flow, to help understand who, what, when, where and how the traffic is flowing. It also detects network anomalies and security vulnerabilities.
TCPDstat	Produces per-protocol breakdown of traffic, for a given libpcap file, like number of packets, average rate and its standard deviation, number of unique source and destination address pairs. It is also useful in getting a high-level view of traffic patterns.

Continued on next page

Name of the NFAT	Description
Ngrep	A pcap-aware tool that allows specifying extended regular or hexadecimal expressions to match against data payloads. It can debug plaintext protocol interactions to identify and analyze anomalous network communications and to store, read and reprocess pcap dump files while looking for specific data patterns.
TCPXtract	Extracts files from network traffic based on file signatures. It can also be used to intercept files transmitted across networks.
SiLK	Supports efficient capture, storage and analysis of network flow data based on CiscoNetFlow. The tool suite, consisting of collection and analysis tools, provides analysts with the means to understand, query, and summarize both recent and historical traffic data in network flow records. SiLK supports network forensics in identifying artifacts of intrusions, vulnerability exploits, worm behavior, etc. SiLK has performance as a key element and manages the large volume of traffic by storing only the security-related information.
TCPReplay	Suite of tools with ability to classify previously captured traffic as client or server, rewrite layer 2, 3 and 4 headers and finally replay the traffic back onto the network. TCPPrep is a multi-pass pcap file pre-processor which determines packets as client or server, TCPRewrite is a pcap file editor which rewrites packet headers, TCPReplay replays pcap files at arbitrary speeds onto the network and TCPBridge bridges two network segments.
P0f	Passive OS fingerprinting by capturing traffic coming from a host to the network. It can also detect the presence of firewall, use of NAT, existence of a load balancer setup, distance to the remote system and its uptime.
Nmap	Utility for network exploration and security auditing. It supports many types of port scans and can be used as on OS fingerprinting tool. It uses raw IP packets in novel ways to determine hosts available on the network, services being offered, operating systems running, firewalls in use and many other characteristics.
Bro	NIDS that passively monitors network traffic for suspicious activity. It detects intrusions by first parsing network traffic to extract its application-level semantics and then executing event-oriented analyzers that compare this activity with patterns deemed troublesome. It is primarily a research platform for IDS, traffic analysis and network forensics.
Snort	Network intrusion prevention/detection system capable of performing packet logging, sniffing and real-time traffic analysis. It can perform protocol analysis, content searching, matching and application-level analysis. It can capture the traffic in libpcap format.

Table C.1 – Description of network security and monitoring (NSM) tools.

Appendix D

Design of *Netfox.Framework*

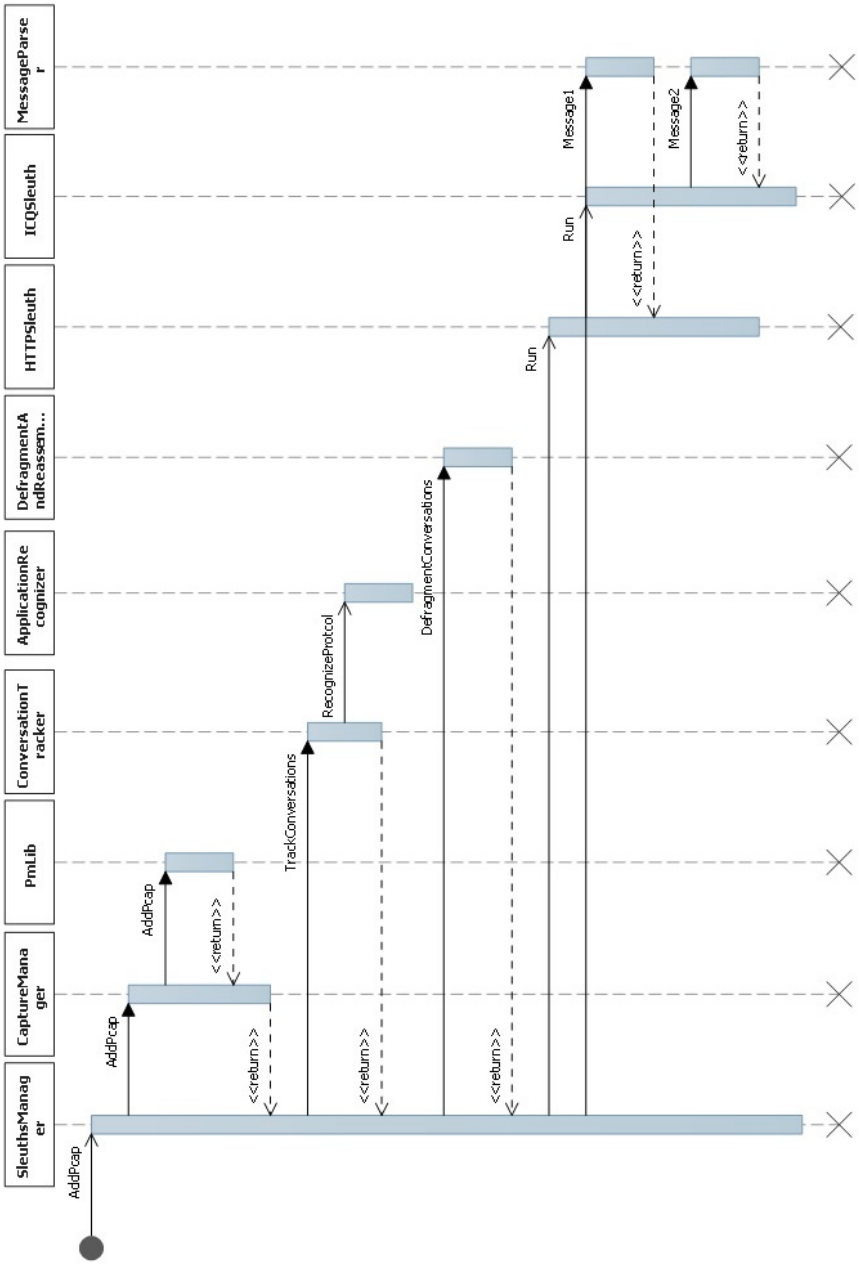


Figure D.1: Sequence diagram describing *Netfox.Framework* in processing of HTTP and ICQ data reconstruction from captured traffic.

Appendix E

NPL

Bits	Data type	Data cast
8	INT8 UINT8	INT8 UINT8 PeekINT8 PeekINT8
16	INT16 UINT16	INT16 UINT16 PeekINT16 PeekINT16
32	INT32 UINT32	INT32 UINT32 PeekINT32 PeekINT32
64	INT64 UINT64	INT64 UINT64 PeekINT64 PeekINT64

Table E.1: This table shows basic data types and casts

Data type	Data cast
String	String
AsciiString	AsciiString
AsciiStringTerm	AsciiStringTerm
Blob	
DnsString	

Table E.2: This table shows custom data types and casts

Data type	**Supported by NPlangCompiler**
AddToProperty	YES
BuildConversation	YES
BuildConversationWithParent	NO
BuildConversationWithProcess	NO
ConditionalLogMessage	NO
Contains	YES
ContainsBin	NO
FormatString	YES
GetProcessID	NO
GetProcessName	NO
IsValueNone	NO
LogMessage	NO
MakeByteArray	NO
NameToHost	NO
PayloadStart	NO
ProxyBuildConversation	NO
ProxyBuildConversationWithParent	NO
StringToNumber	YES
ToBitString	NO
ToWideChar	NO
TypeCast	NO
ValidateTCPCheckSum	NO

Table E.3: This table shows custom data types and casts

```
//[RegisterAfter(TCPPayload.HTTP, YMSG, 5050)]
//[RegisterAfter(PayloadHeader.LLC, YMSG, YMSG)]
Protocol YMSG = FormatString("Type = %s (%d), status = %s (%d)",
                    YMSGTypes(Type),Type, YMSGHEADStatus(Status),Status)
{
  AsciiString(4) YMSGconst;
  UINT16 Version;
  UINT16 VendorID;
  [Post.Properties.Length = Length]
  UINT16 Length;
  UINT16 Type = FormatString("%s (%#04x)", YMSGTypes(Type), Type);
  UINT32 Status = FormatString("%s (%#04x)", YMSGHEADStatus(this), this);
  UINT32 Session;

/* [BuildConversationWithParent(Session)]
StartPayload
  [YMSGPayload = Blob(FrameData, FrameOffset, Length),
  DataFieldFrameLength = frameOffset + Length,
  PayloadStart (
  NetworkDirection, //direction
  0, // id
  0, // sequence token
  0, // next sequence token
  Length, // total payload length
  !Property.TCPContinuation, // is first
  (TCP.Flags.Push || TCP.Flags.Fin || TCP.Flags.Urgent),//is last
  RssmblyIndStartBit + RssmblyIndEndBit // Properties...
  )]*/

  [HeaderOffset = FrameOffset]
  TVs tvs;
  status = "%s (%d)", YMSGTypes(Type),Type,YMSGHEADStatus(Status),Status)]
  switch{
    case (FrameLength - FrameOffset - 1 > 0) && (HeaderOffset <=
        FrameOffset):
      [MultiYMSG = "YES"]
      StringTerm(0, "YMSG", 1, 0, 0 ) blank0 = FormatString("Optional
          stuffing between multiple YMSG messages", ymsg);
      YMSG ymsg;
    case (FrameLength - FrameOffset > 0) && (HeaderOffset <= FrameOffset):
      BLOB(FrameLength - FrameOffset) blank1 = FormatString("Optional
          terminator of YMSG message", this);
  }
}
```

Listing E.1: Example of not complete NPL description of YMSG IM protocol. Commented parts are not supported by NPlangCompiler.

Appendix F

Description of XML log used as an interphase between Sleuths and ContextBrowser

```
<?xml version="1.0" encoding="utf-8"?>
<log>
  <user guid="192.168.2.101">
    <protocol val="ICQ">
      <event validity="valid">
        <src version="IPv4" port="49874">192.168.2.101</src>
        <dst version="IPv4" port="5190">205.188.10.251</dst>
        <conversationPart type="statusChange" timeStamp="2013-08-08T02:06:19.518767"
        frameNumber="86" flowDirection="up" statusType="online" />
        <conversationPart type="contactList" timeStamp="2013-08-08T02:06:39.318927"
        frameNumber="113" flowDirection="down">
          <group id="0" name="">
            <contact id="3" firstName="277264821"
              nick="Contact1" />
          </group>
          <group id="1" name="rename_group" />
          <group id="3" name="friends_group">
            <contact id="9327" firstName="647175775"
              nick="Contact2" />
            <contact id="2" firstName="284569266"
              nick="Contact3" />
            <contact id="29425" firstName="312345170"
              nick="Contact4" />
          </group>
        </conversationPart>
        <conversationPart type="message"
        timeStamp="2013-08-08T02:11:18.357794" frameNumber="791" flowDirection="up">
          <![CDATA[<HTML><BODY dir="ltr"><FONT size="2">test</FONT></BODY></HTML>]]>
          <receiver>310451170</receiver>
        </conversationPart>
        <conversationPart type="contactListChange"
        timeStamp="2013-08-08T02:32:11.914733" frameNumber="1528" flowDirection="up"
        contactListChangeAction="add">
          <contact id="31432670" />
        </conversationPart>
        <conversationPart type="authorizationReply"
        timeStamp="2013-08-08T02:32:55.885157"
          frameNumber="1543" flowDirection="down" sender="310451170"
          authorizationStatus="permit" />
    </protocol>
  </user>
</log>
```

Listing F.1: Example of IMSleuths's xml log used as input in ContextBrowser.

```
<?xml version="1.0" encoding="utf-16"?>
<!ELEMENT group (contact)*>
<!ATTLIST group
  id CDATA #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT contact (#PCDATA)>
<!ATTLIST contact
  nick CDATA #REQUIRED
  id CDATA #REQUIRED
  firstName CDATA #IMPLIED
  group CDATA #IMPLIED
  lastName CDATA #IMPLIED
  >
<!ELEMENT receiver (#PCDATA)>

<!ELEMENT clientId EMPTY>
<!ATTLIST clientId id CDATA #REQUIRED>

<!ELEMENT conversationPart (#PCDATA|contact|group|receiver)*>
<!ATTLIST conversationPart
  flowDirection (up|down) #REQUIRED
  frameNumber CDATA #REQUIRED
  timeStamp CDATA #REQUIRED
  type (message|statusChange|authorizationRequest|authorizationReply|contactListChange|
  contactDetails|fileTransfer|contactList) #REQUIRED
  authorizationStatus (permit|deny|unknown) #IMPLIED
  receiver CDATA #IMPLIED
  sender CDATA #IMPLIED
  source CDATA #IMPLIED
  statusType (online|offline|away|busy|notavaible|log_on|log_off|free_for_chat|unknown) #IMPLIED
  charset CDATA #IMPLIED
  contactListChangeAction (add|remove|block|unblock|edit) #IMPLIED
  exportedFileName CDATA #IMPLIED
  fileName CDATA #IMPLIED
  size CDATA #IMPLIED
  >

<!ELEMENT src (#PCDATA)>
<!ATTLIST src
  port CDATA #REQUIRED
  version CDATA #REQUIRED>

<!ELEMENT dst (#PCDATA)>
<!ATTLIST dst
  port CDATA #REQUIRED
  version CDATA #REQUIRED>

<!ELEMENT event (src,dst,(clientId)?,(conversationPart)*)>

<!ELEMENT protocol (event)+>
<!ATTLIST protocol val CDATA #REQUIRED>
<!ELEMENT user (protocol)+>
<!ATTLIST user guid CDATA #REQUIRED>
<!ELEMENT log (user)+>
```

Listing F.2: XML schema used as a interphase between IMSleuths and ContextBrowser.